



• 第4章 软件架构的风格与模式

王璐璐 wanglulu@seu.edu.cn

廖力 lliao@seu.edu.cn



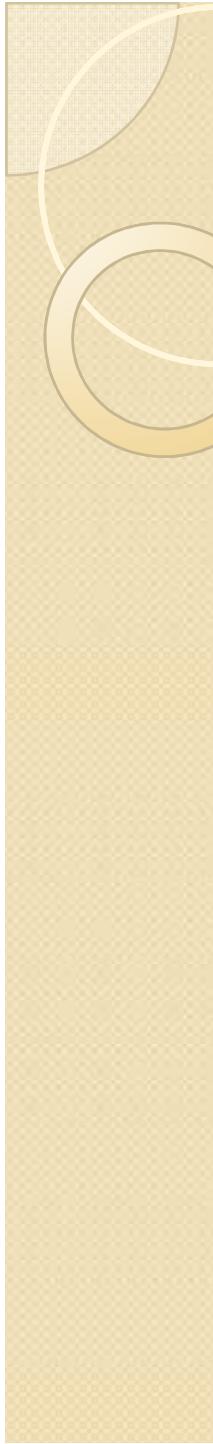
第4章 软件架构的风格与模式

- 4. 1软件架构风格定义
- 4. 2软件架构风格的分类
- 4. 3典型的软件架构风格
- 4. 4软件架构模式



4.1 软件架构风格的定义

- 软件架构风格 (software architecture style)，又称软件架构惯用模式 (software architecture idiomatic paradigm)，是描述某一特定应用领域中系统组织方式的惯用模式，作为“可复用的组织模式和习语”，为设计人员的交流提供了公共的术语空间，促进了设计复用与代码复用。



4.1 软件架构风格的定义

- 架构风格的基本属性
 - 设计元素的词汇表（包括组件、连接件的类型以及数据元素，例如：管道，过滤器，对象，服务等）
 - 配置规则：决定元素组合的拓扑约束
 - 例如限制某一风格中的组件至多与其它两个组件相连。
 - 元素组合的语义解释以及使用某种风格构建的系统的相关分析



4.1 软件架构风格的定义

- 使用架构风格的优势
 - 它极大地促进了设计的重用性和代码的重用性，并且使得系统的组织结构易被理解。
 - 即使没有给出具体实现细节，依然可以通过“客户/服务器”架构风格大致推测出系统的组成结构和工作方式。
 - 使用标准的架构风格可较好地支持系统内部的互操作性以及针对特定风格的分析
 - 如：“管道/过滤器”风格可用来分析调度、吞吐量、延迟，和死锁等问题。



4.2 软件架构风格的分类

- (1) 数据流风格：批处理序列、管道/过滤器；
- (2) 调用/返回风格：主程序/子程序、面向对象风格、层次结构；
- (3) 独立组件风格：进程通讯、事件系统；
- (4) 虚拟机风格：解释器、基于规则的系统；
- (5) 仓库风格：数据库系统、超文本系统、黑板系统等。

4.3 典型的软件架构风格

- 1. 管道过滤器风格
- 2. 主程序/子程序风格
- 3. 面向对象风格
- 4. 层次化风格
- 5. 事件驱动风格
- 6. 解释器风格
- 7. 基于规则的系统风格
- 8. 仓库风格
- 9. 黑板系统风格
- 10. C2风格
- 11. 客户机/服务器风格
- 12. 浏览器/服务器风格
- 13. 平台/插件风格
- 14. 面向Agent风格
- 15. 面向方面软件架构风格
- 16. 面向服务架构风格
- 17. 正交架构风格
- 18. 异构风格
- 19. 基于层次消息总线的架构风格
- 20. 模型-视图-控制器风格

4.3.1 管道过滤器风格

- 基本思想
 - 在管道过滤器风格下，每个功能模块都有一组输入和输出。
 - 功能模块称作过滤器（filters）；功能模块间的连接可以看作输入、输出数据流之间的通路，所以称作管道（pipes）。
 - 管道—过滤器风格的特性之一在于过滤器的相对独立性，即过滤器独立完成自身功能，相互之间无需进行状态交互。

4.3.1 管道-过滤器风格

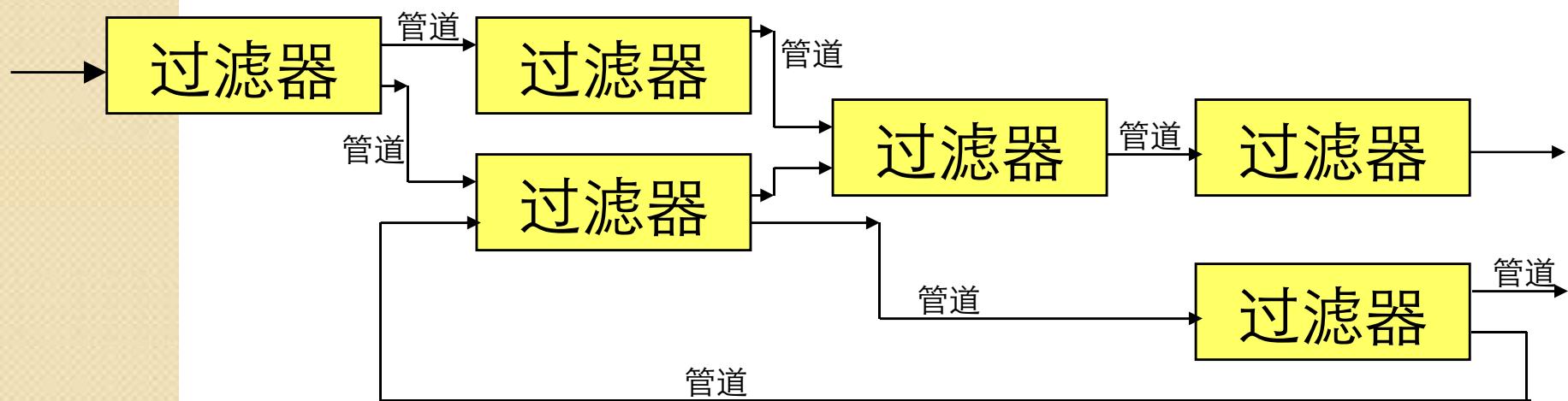


图4.1 管道 – 过滤器风格的体系结构



4.3.1 管道-过滤器风格

- 过滤器是独立运行的组件
 - 非临近的过滤器之间不共享状态
 - 过滤器自身无状态
- 过滤器对其处理上下连接的过滤器“无知”
 - 对相邻的过滤器不施加任何限制
- 结果的正确性不依赖于各个过滤器运行的先后次序
 - 各过滤器在输入具备后完成自己的计算。完整的计算过程包含在过滤器之间的拓扑结构中。



管道-过滤器风格实例

- 在 Unix 和 Dos 中，允许在命令中出现用竖线字符 “|” 分开的多个命令，将符号 “|” 之前的命令的输出，作为 “|” 之后命令的输入，这就是 “管道功能”，竖线字符 “|” 是管道操作符。
- 例如： Unix shell 中：
 - `cat file|grep xyz|sort|uniq>out`
 - 即找到含xyz的行，排序、去掉相同的行，最后输出
- 例如： DOS 中：
 - 命令 `dir | more` 使得当前目录列表在屏幕上逐屏显示。
 - `dir` 的输出是整个目录列表，它不出现在屏幕上而是由于符号 “|” 的规定，成为下一个命令 `more` 的输入，`more` 命令则将其输入，`more` 命令则将其输入一屏一屏地显示，成为命令行的输出。

管道-过滤器风格 实例

- dir | more

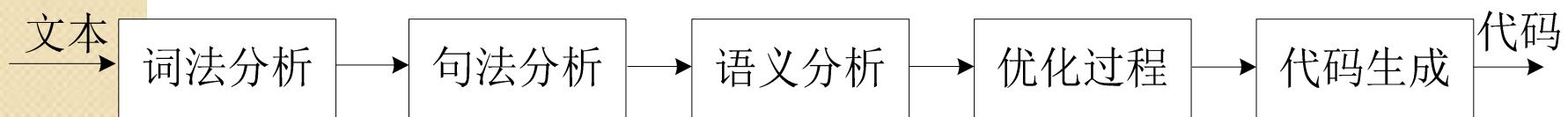
```
C:\>cd windows\system32
C:\>dir | more
驱动器 c 中的卷没有标签。
卷的序列号是 30CC-6AD3

C:\windows\system32 的目录

2008-10-17 19:23    <DIR>          .
2008-10-17 19:23    <DIR>          ..
2008-10-09 10:35            378 $winnt$.inf
2008-10-09 18:07    <DIR>          1025
2008-10-09 18:07    <DIR>          1028
2008-10-09 18:07    <DIR>          1031
2008-10-09 18:08    <DIR>          1033
2008-10-09 18:07    <DIR>          1037
2008-10-09 18:07    <DIR>          1041
2008-10-09 18:07    <DIR>          1042
2008-10-09 18:07    <DIR>          1054
2008-04-14 20:00            2,151 12520437.cpx
2008-04-14 20:00            2,233 12520850.cpx
2008-10-09 18:09    <DIR>          2052
2008-10-09 18:07    <DIR>          3076
2008-10-09 18:07    <DIR>          3com_dmi
2008-04-14 20:00            100,352 6to4svc.dll
2008-04-14 20:00              1,460 a15.tbl
2008-04-14 20:00             44,370 a234.tbl
-- More --
```

管道-过滤器风格实例

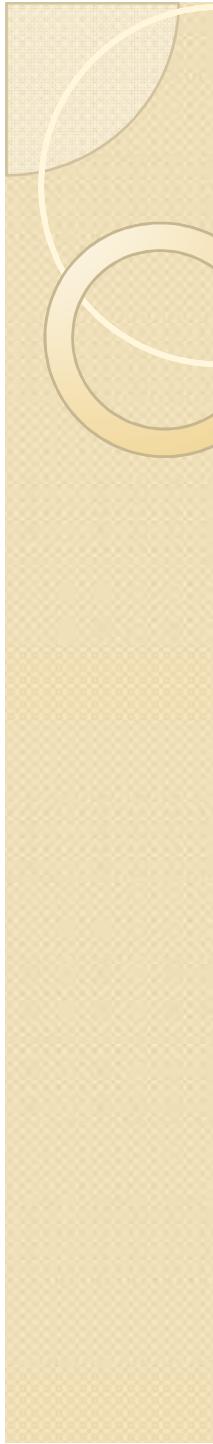
- 实例：
 - 传统的编译器，一个阶段的输出是另一个阶段的输入（包括词法分析、语法分析、语义分析和代码生成）





管道-过滤器风格优点

- (1) 由于每个组件行为不受其他组件的影响，整个系统的行为易于理解。



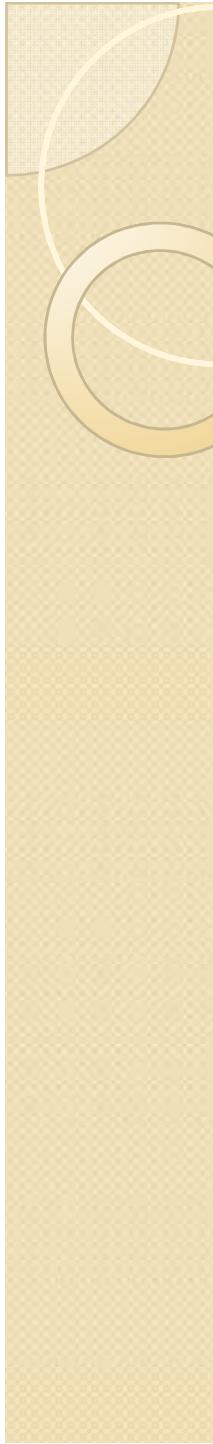
管道-过滤器风格优点

- (2) 管道-过滤器风格支持功能模块的复用
 - 任何两个过滤器，只要它们之间传送的数据遵守共同的规约，就可以相连接。每个过滤器都有自己独立的输入输出接口，如果过滤器间传输的数据遵守其规约，只要用管道将它们连接就可以正常工作。



管道-过滤器风格优点

- (3) 基于管道-过滤器风格的系统具有较强的可维护性和可扩展性。
 - 旧的过滤器可以被替代，新的过滤器可以添加到已有的系统上。软件的易于维护和升级是衡量软件系统质量的重要指标之一，在管道-过滤器模型中，只要遵守输入输出数据规约，任何一个过滤器都可以被另一个新的过滤器代替，同时为增强程序功能，可以添加新的过滤器。这样，系统的可维护性和可升级性得到了保证。



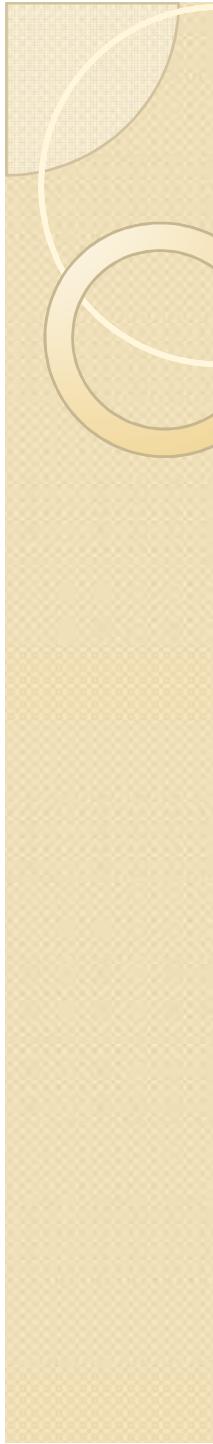
管道-过滤器风格优点

- (4) 支持一些特定的分析，如吞吐量计算和死锁检测等。
 - 利用管道-过滤器风格的视图，可以很容易的得到系统的资源使用和请求的状态图。然后，根据操作系统原理等相关理论中的死锁检测方法就可以分析出系统目前所处的状态，是否存在死锁可能及如何消除死锁等问题。



管道-过滤器风格优点

- (5) 管道-过滤器风格具有并发性
 - 每个过滤器作为一个单独的执行任务，可以与其它过滤器并发执行。过滤器的执行是独立的，不依赖于其它过滤器的。在实际运行时，可以将存在并发可能的多个过滤器看作多个并发的任务并行执行，从而大大提高系统的整体效率，加快处理速度。



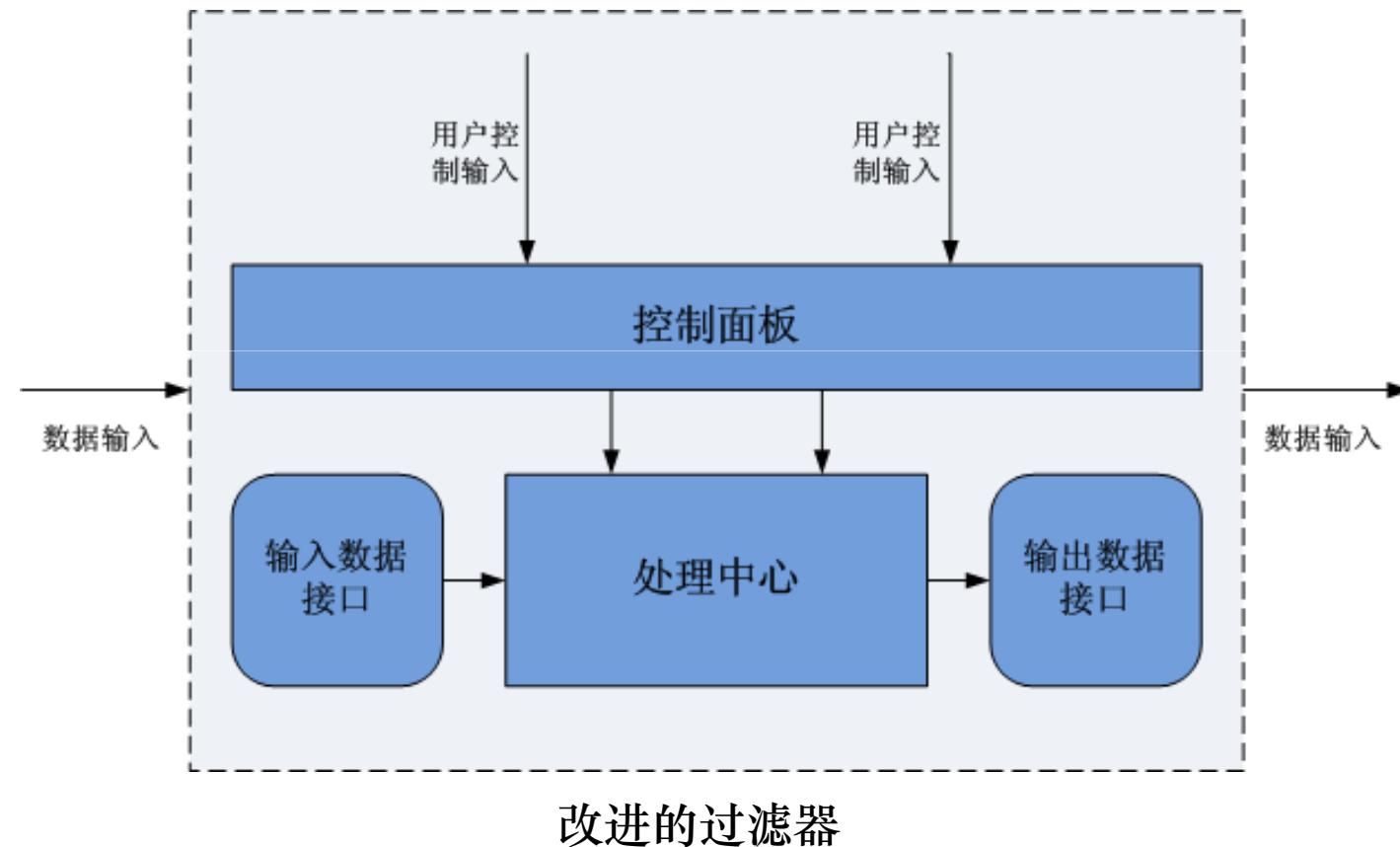
管道-过滤器风格不足

- (1) 管道-过滤器风格往往导致系统处理过程的成批操作。
- (2) 根据实际设计的需要，设计者也需要对数据传输进行特定的处理（如为了防止数据泄漏而采取加密等手段，或者使用了底层公共命名），导致过滤器必须对输入、输出管道中的数据流进行解析或反解析，增加了过滤器具体实现的复杂性。

管道-过滤器风格不足

- (3) 交互式处理能力弱
 - 管道-过滤器模型适于数据流的处理和变换，在系统中导区。涉及较过滤器模型。数据，系统自己读入，这样，多为频繁的都储输出数据，要其实现过滤器可以与用户交互过滤器共享数据操作，使得外部可以不适合模型。这些或者整个用户过滤器由以上的用户操作对相应的缺点，接口，使得外部可以这种数据是由系统要操作对以上的用户进行控制。

管道-过滤器风格不足

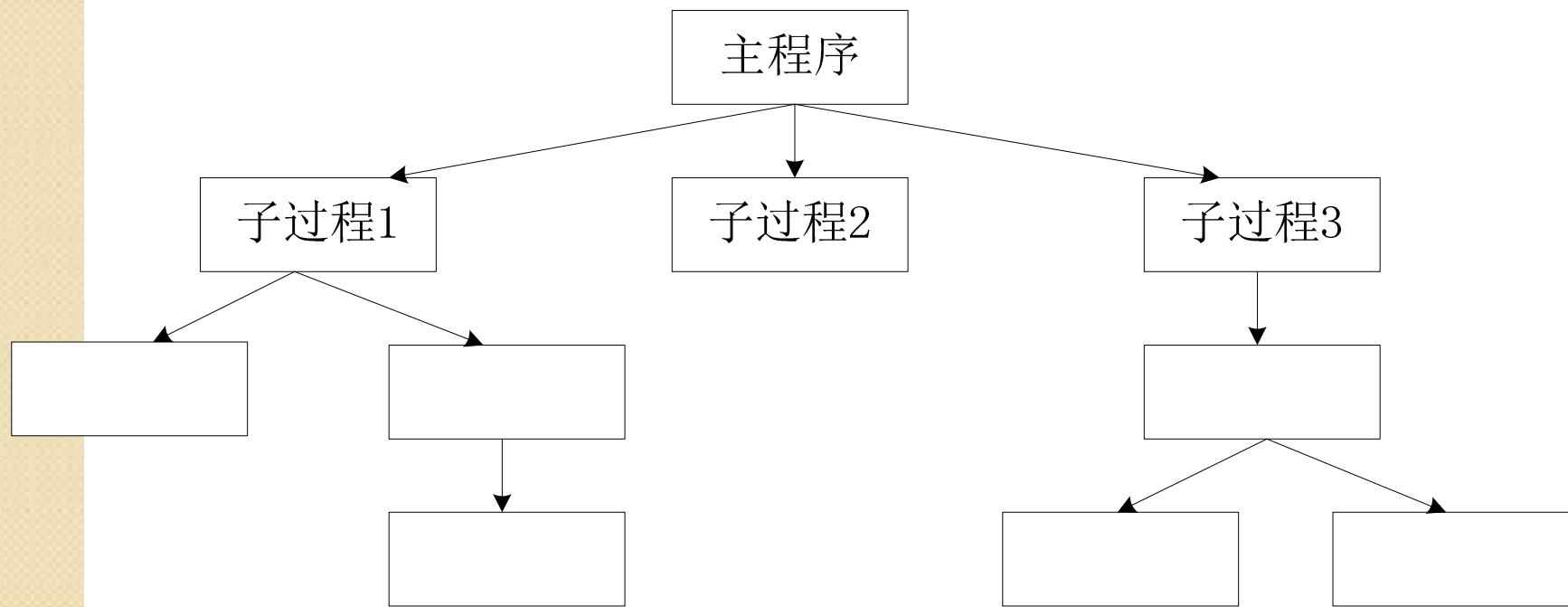


4.3.2 主程序/子程序风格

- 20世纪70年代，结构化程序设计方式出现，主程序/子程序结构(Main Program/Subroutines) 成为结构化设计的一种典型风格。
- 该架构风格从功能的观点设计系统，通过逐步分解和逐步细化得到系统架构，即将大系统分解为若干模块(模块化)，主程序调用这些模块实现完整的系统功能，因此主程序的正确性依赖于它所调用的子程序的正确性。

4.3.2 主程序/子程序风格

- 其中，组件为主程序和子程序，连接件为调用-返回机制，拓扑结构为层次化结构。



4.3.2 主程序/子程序风格

- 优缺点分析

- 优点：（1）具有很高的数据访问效率，因为计算共享同一个存储区。（2）不同的计算功能被划分在不同的模块中。
- 缺点：该方案在处理变更的能力上有许多严重的缺陷：（1）对数据存储格式的变化将会影响几乎所有的模块。（2）对处理流程的改变与系统功能的增强也很难适应，依赖于控制模块内部的调用次序。（3）这种分解方案难以支持有效的复用。

4.3.2 主程序/子程序风格

- 应用实例

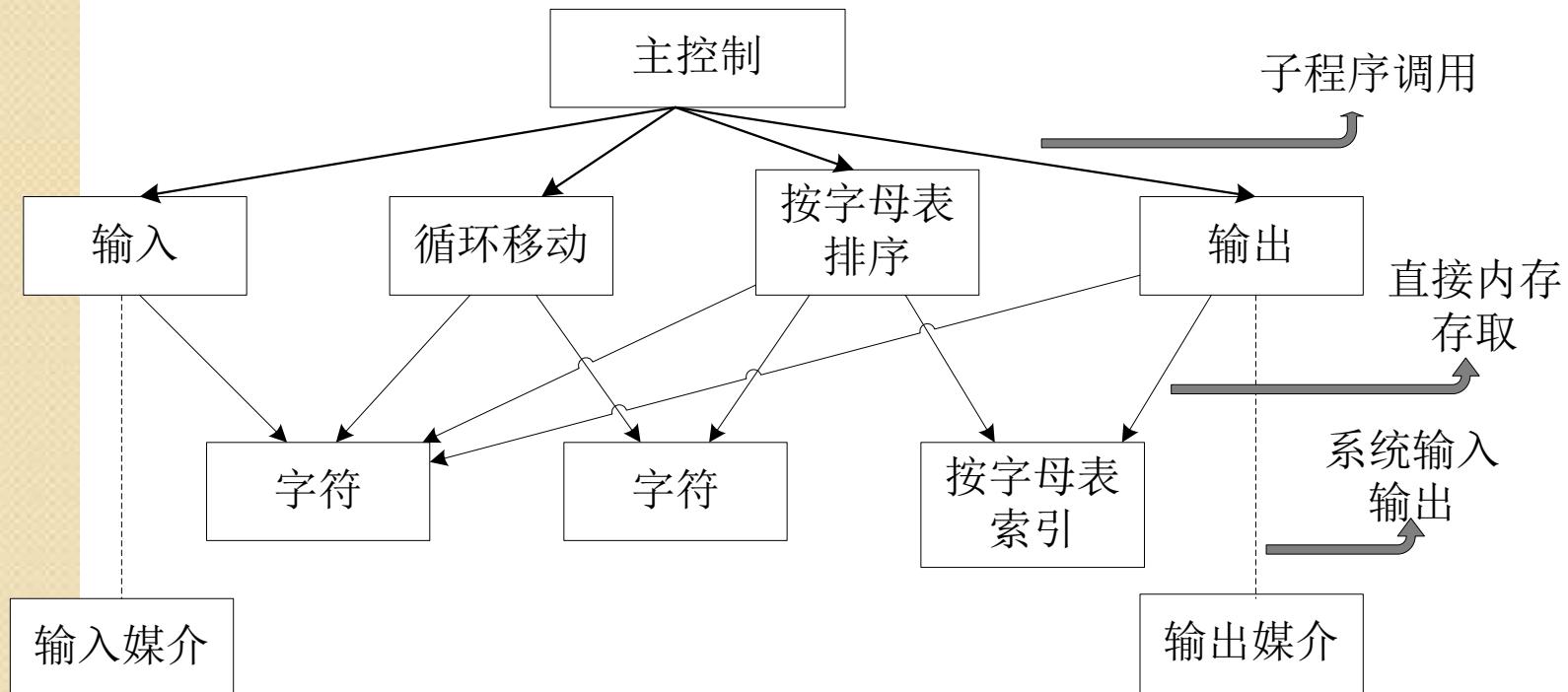


图4.6 KWIC (Key Word in Context) 解决方案

4.3.3 面向对象风格

- 面向对象 (Object-Oriented) 方法是80年代初期提出的一种新型的程序设计方法，它彻底改变了过去数据流、事物流分析自然抽分方法，对问题域进行面向对象分析的完整体系。该架构风格从现实世界中客观存在的事物出发，强调直接以问题域中的事物为中心，根据这些事物的本质特征，将其抽象为系统中的对象，并作为系统中的基本构成单位。



4.3.3 面向对象风格

- 面向对象组织结构有两个重要特点：
 - (1)对象负责维护其表示的完整性（通常通过保持其表示上的一些不变式来实现的）；
 - (2)对象的表示对其他对象而言是隐蔽的。抽象数据类型的使用，以及面向对象系统的使用已经非常普遍。

4.3.3 面向对象风格

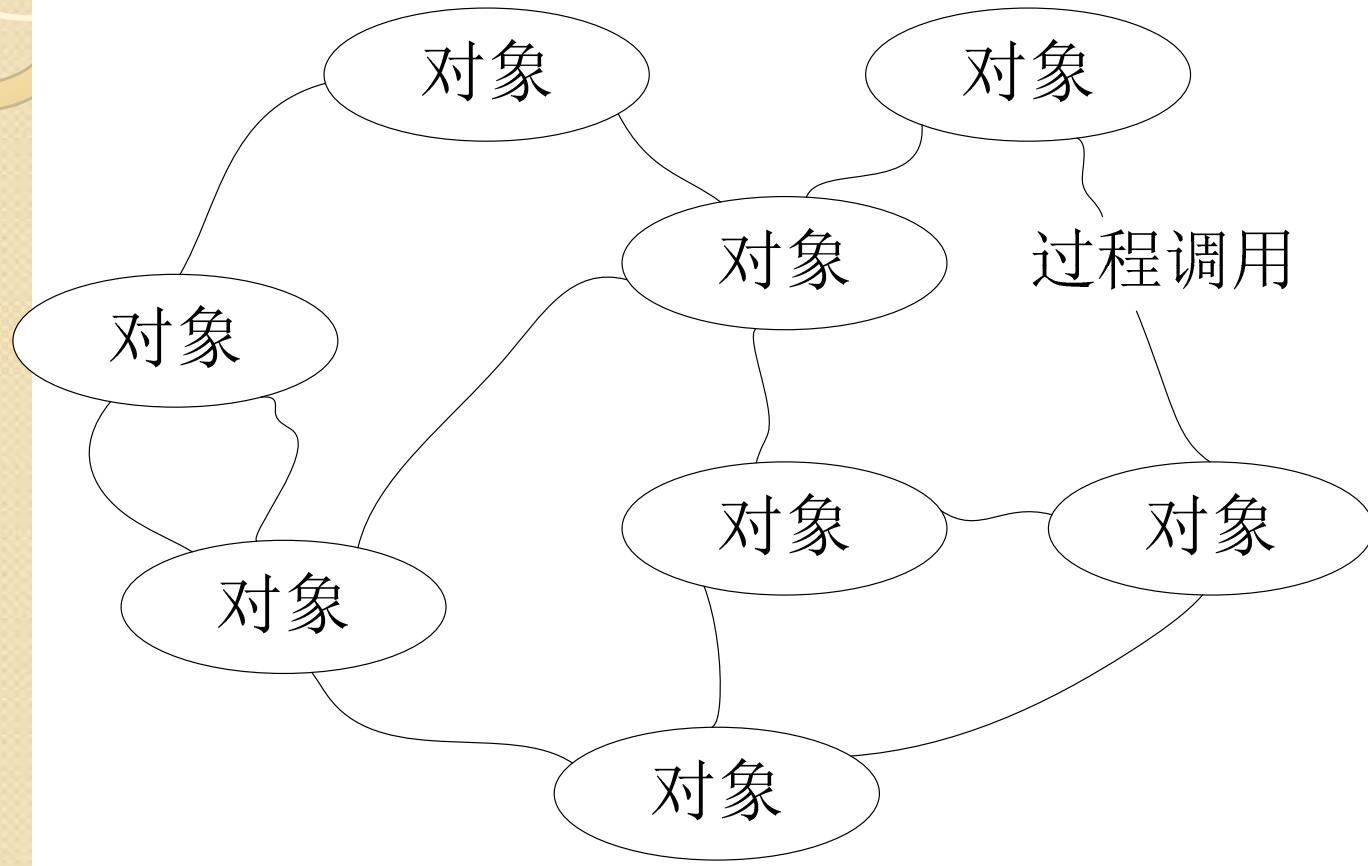


图4-7面向对象风格



4.3.3 面向对象风格

- 优点：
 - (1) 对象隐藏了其实现细节，所以可以在不影响其它对象的情况下改变对象的实现，不仅使得对象的使用变得简单、方便，而且具有很高的安全性和可靠性。
 - (2) 设计者可将一些数据存取操作的问题分解成一些交互的代理程序的集合。

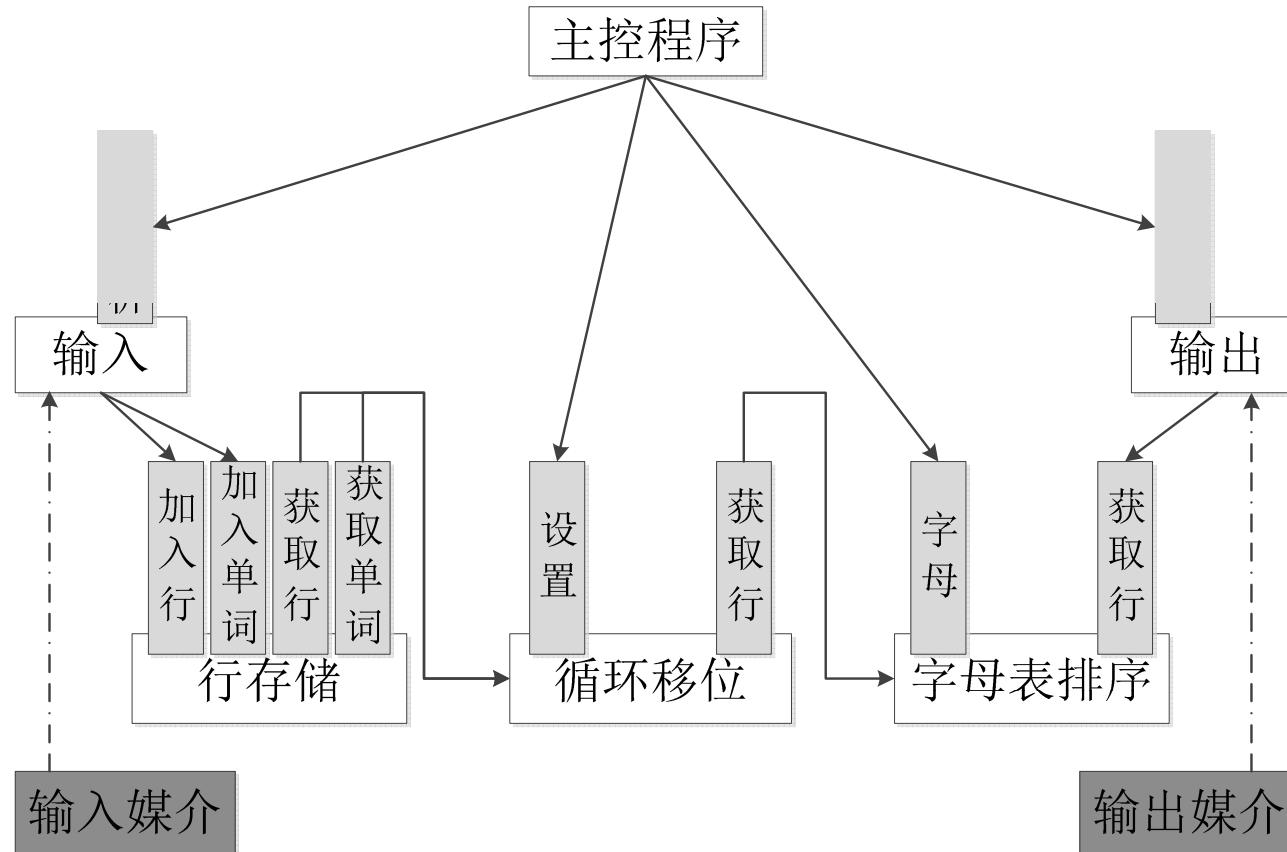


4.3.3 面向对象风格

- 缺点
 - 当一个对象和其它对象通过过程调用等方式进行交互时，必须知道其它对象的标识。无论何时改变对象的标识，都必须修改所有显式调用它的其它对象，并消除由此带来的一些副作用。

4.3.3 面向对象风格

- 应用实例：KWIC检索系统



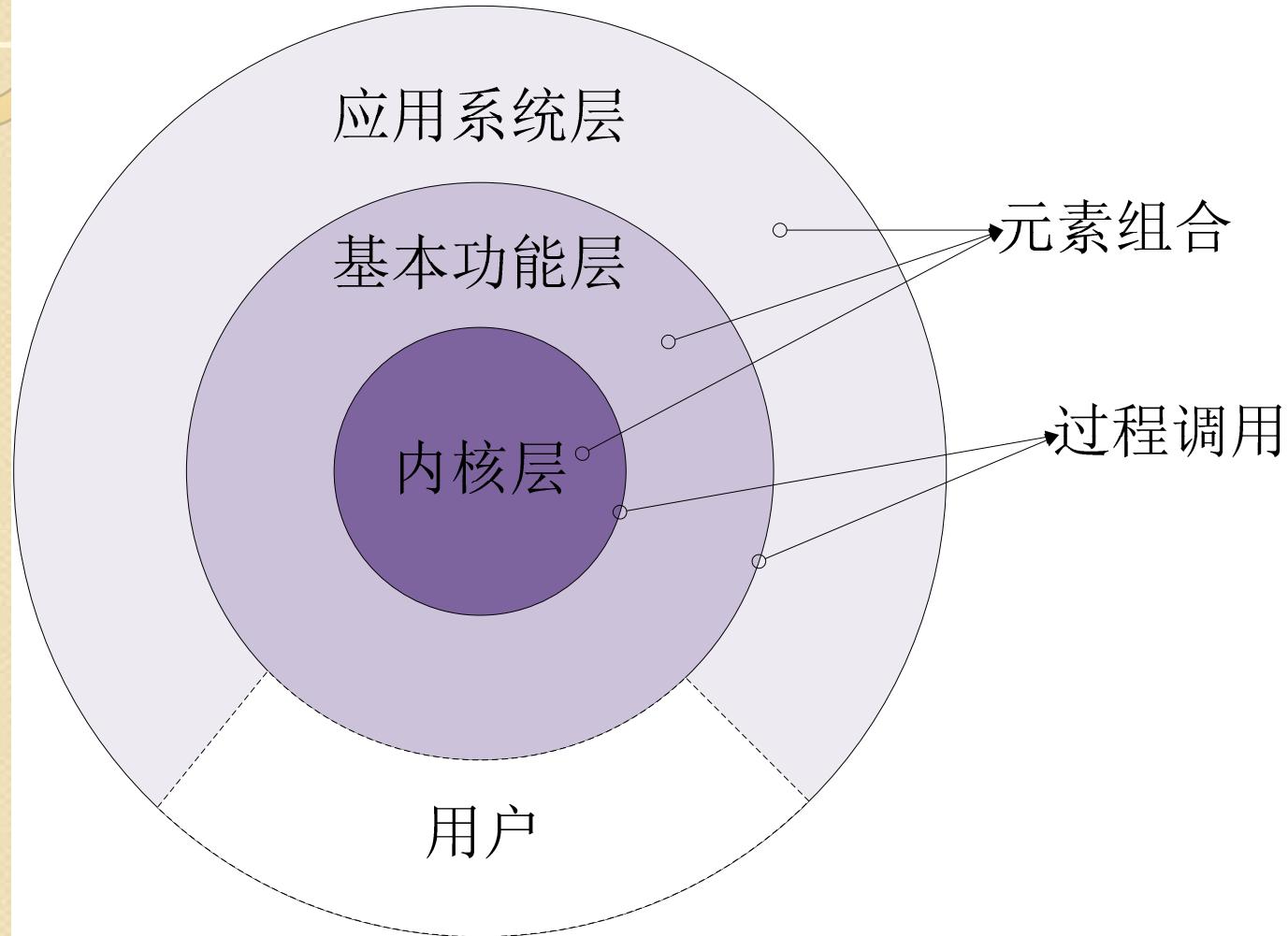
图例: □ 模块（对象） ■ 公共接口 ■ 输入/输出媒介
→ 方法调用 - - - → 系统输入/输出



4.3.4 层次化风格

- 基本思想
 - 在分层系统（Layered System）中，系统被组织成若干个层次，每个层次由一系列组件组成；层次之间存在接口，通过接口形成call/return的关系——下层组件向上层组件提供服务，上层组件被看作是下层组件的客户端。
- 层次化早已经成为一种复杂系统设计的普遍性原则。

4.3.4 层次化风格





4.3.4 层次化风格

- 优点：
 - (1) 支持基于可增加抽象层的设计，允许将一个复杂问题分解成一个增量步骤序列的实现。
 - (2) 支持扩展。每一层的改变最多只影响相邻层。
 - (3) 支持重用。只要给相邻层提供相同的接口，它允许系统中同一层的不同实现相互交换使用。

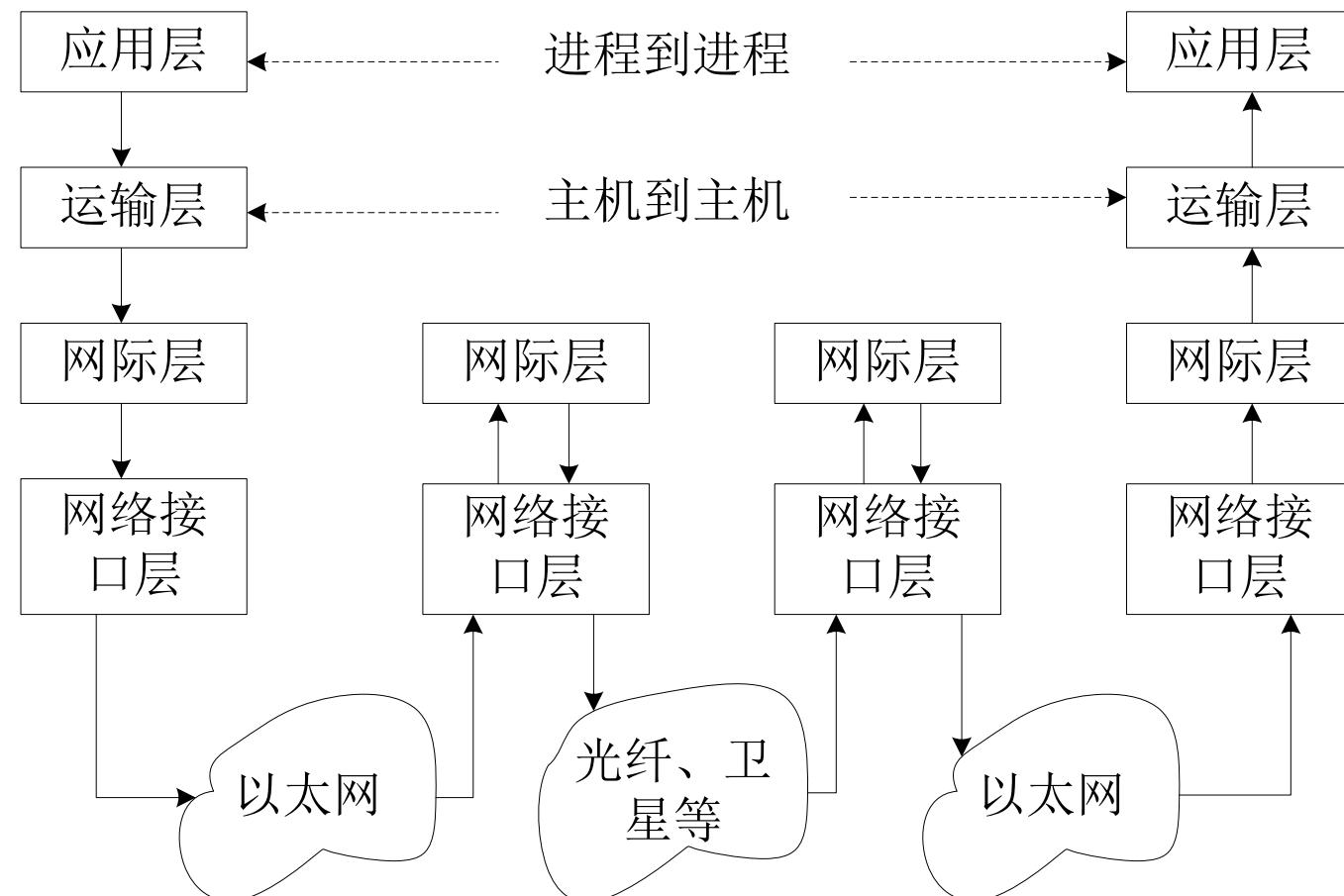


4.3.4 层次化风格

- 缺点
 - (1) 不是所有系统都容易用这种模式来构建；
 - (2) 定义一个合适的抽象层次可能会非常困难，特别是对于标准化的层次模型。
 - 例如，实际的通信协议体就很难映射到ISO框架中，因为其中许多协议跨多个层。

4.3.4 层次化风格

- 应用实例：TCP/IP分层通信协议



4.3.5 事件驱动风格

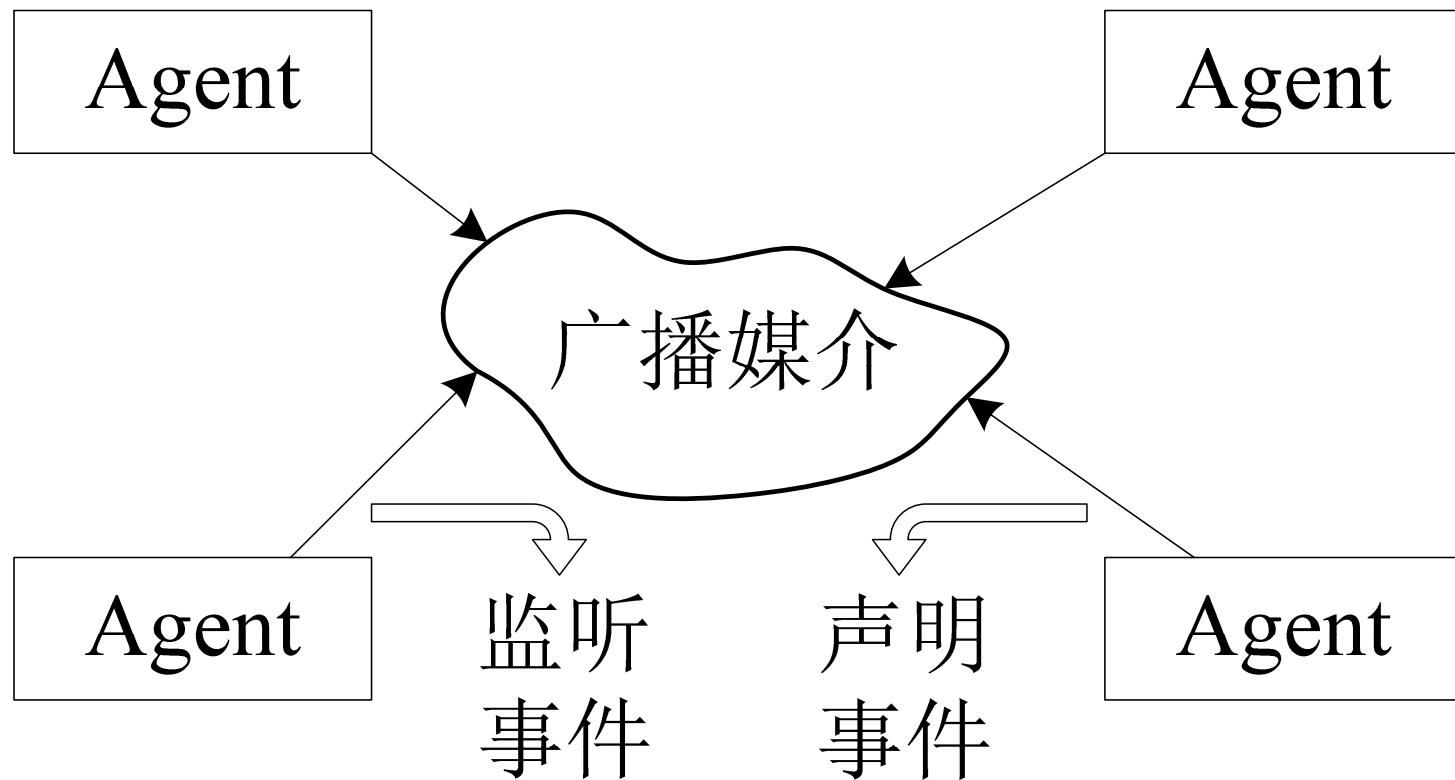
- 事件驱动(Event-based)的风格起源于基于角色(Role-based)的系统、约束满足性检查、后台程序和包交换网络。
- 事件驱动风格中，不直接调用一个过程，而是发布或广播一个或多个事件。系统中的其它组件通过注册与一个事件关联起来的过程，来表示对某一个事件感兴趣。当这个事件发生时，系统本身会调用所有注册了这个事件的过程。这样一个事件的激发会导致其它模块中过程的隐式调用。

4.3.5 事件驱动风格

- 特点：

- 事件发布者不知道哪些组件会受到事件的影响；组件不能对事件的处理顺序，或者事件发生后的处理结果做任何假设。
- 从架构上来说，事件驱动系统的组件提供了一个过程集合和一组事件。
- 过程可以使用显示的方法进行调用，也可以由组件在系统事件中注册。当触发事件时，会自动引发这些过程的调用。
- 连接件既可以是显示过程调用，也可以是一种绑定事件声明和过程调用的手段。

4.3.5 事件驱动风格





4.3.5 事件驱动风格

- 优点
 - (1) 事件声明者不需要知道哪些组件会影响事件，组件之间关联较弱。
 - (2) 提高软件复用能力。只要在系统事件中注册组件的过程，就可以将该组件集成到系统中。
 - (3) 系统便于升级。只要组件名和事件中所注册的过程名保持不变，原有组件就可以被新组件替代。



4.3.5 事件驱动风格

- 缺点
 - (1) 组件放弃了对计算的控制权，完全由系统来决定。
 - (2) 存在数据交换问题。
 - (3) 该风格中，正确性验证成为一个问题。

4.3.5 事件驱动风格

- 应用实例：Field系统

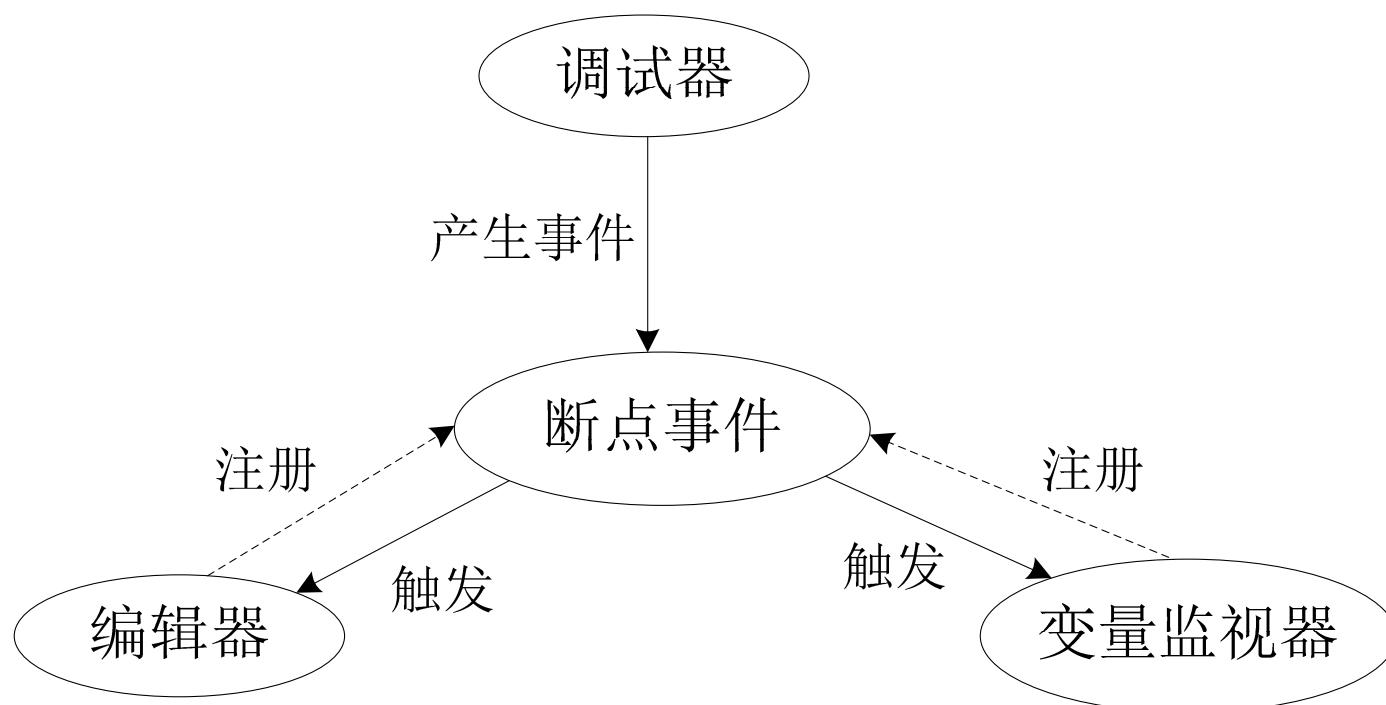
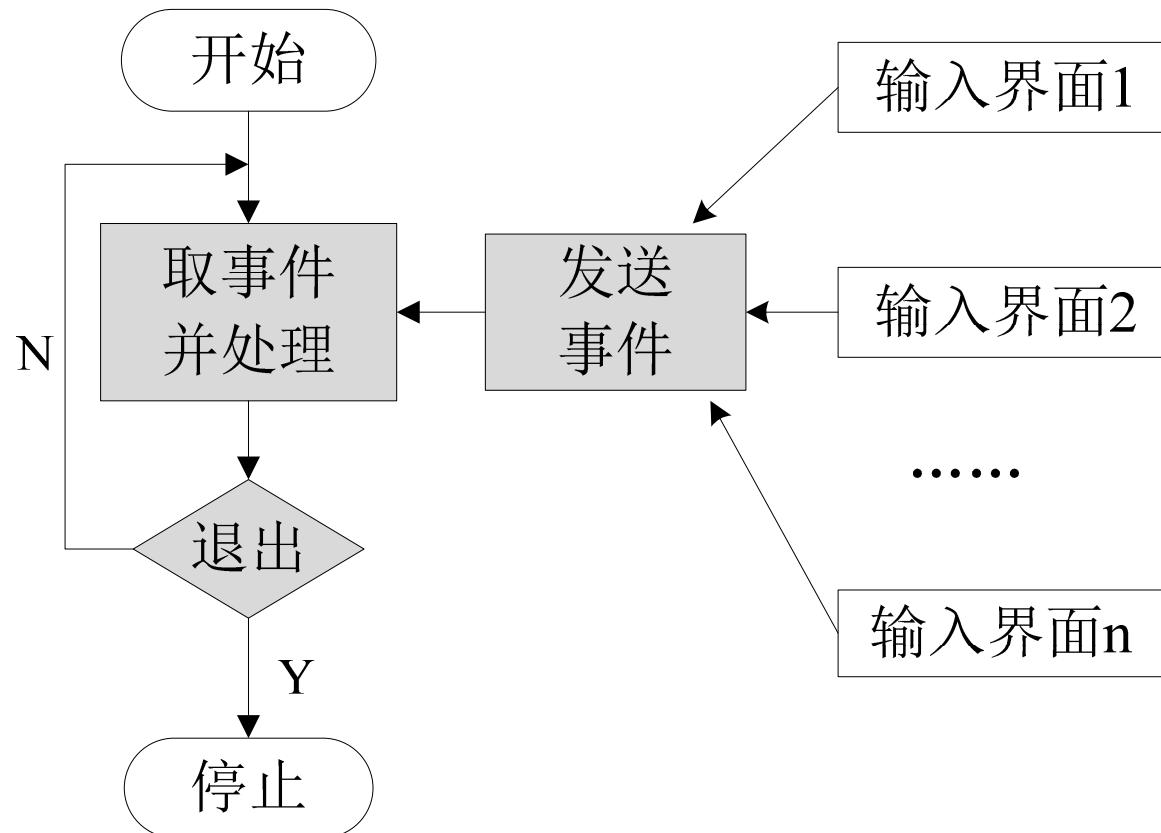


图4.12 调试器中的断点处理

4.3.5 事件驱动风格

- 应用实例：Win32 GUI程序



4.3.6 解释器风格

- 基本思想
 - 解释器(Interpreter) 是一个用来执行其他程序的程序，它针对不同的硬件平台实现了一个虚拟机，将高抽象层次的程序翻译为低抽象层次所能理解的指令，以弥合程序语义所期望的与硬件提供的计算引擎之间的差距。



图4.14解释执行过程

4.3.6 解释器风格

- 组成部分

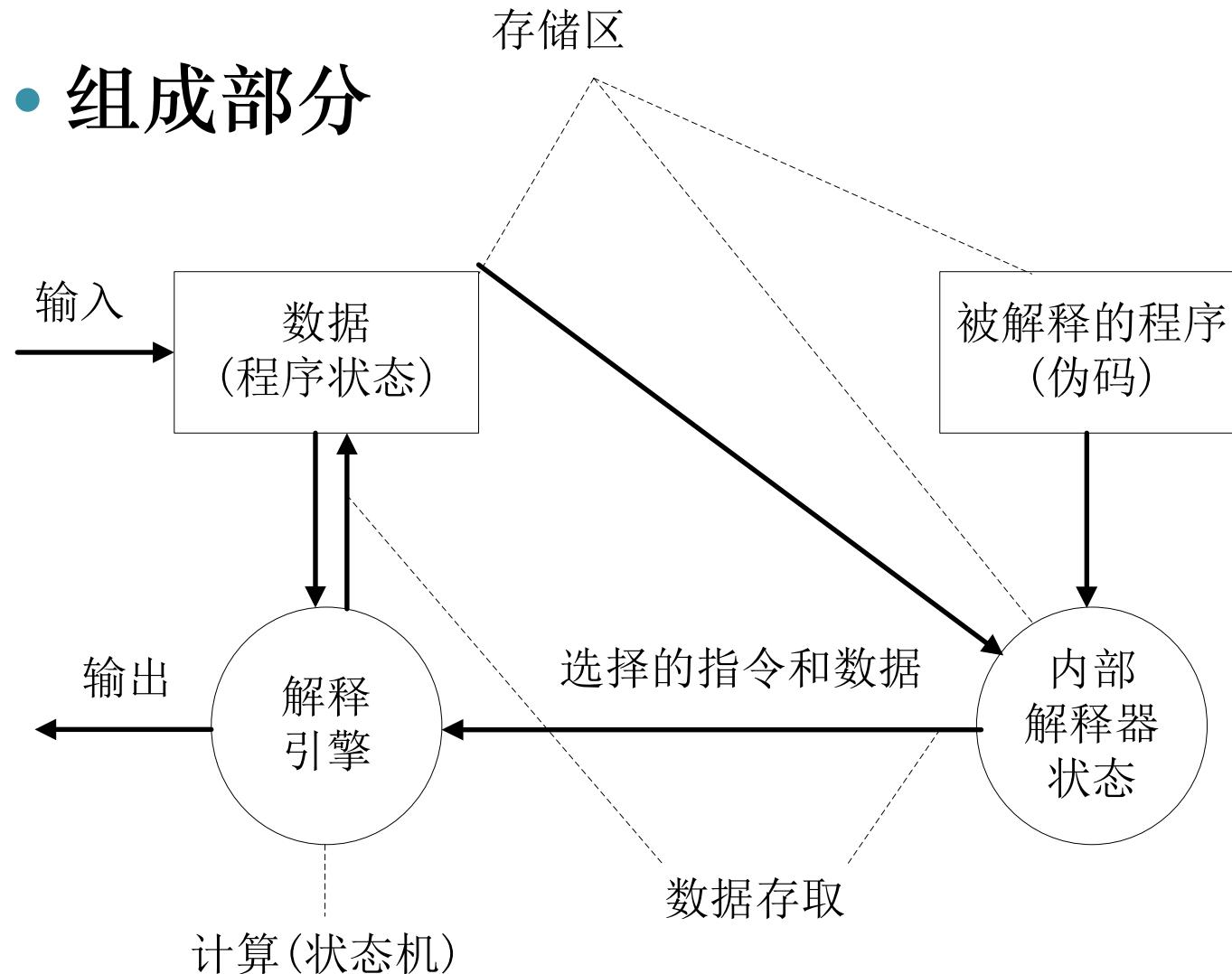


图4.15解释器风格



4.3.6 解释器风格

- 优点
 - (1) 它有利于实现程序的可移植性和语言的跨平台能力；
 - (2) 可以对未来的硬件进行模拟和仿真，能够降低测试所带来的复杂性和昂贵花费
- 缺点
 - 额外的间接层次导致了系统性能的下降，如在不引入JIT (Just In Time) 技术的情况下，Java应用程序的运行速度相当慢。

4.3.6 解释器风格

- 应用实例
 - JVM

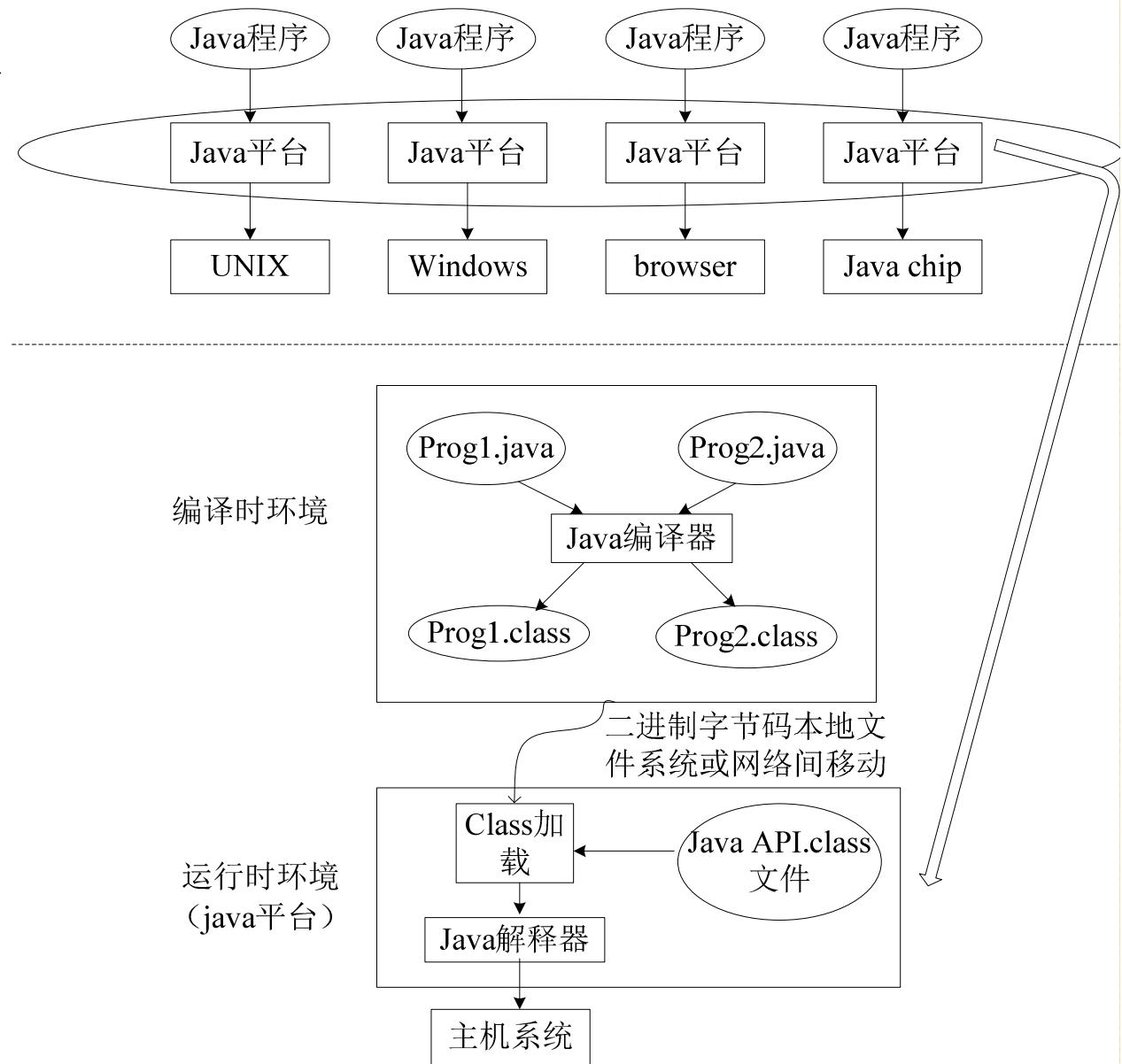


图4.16基于JVM的
Java程序执行过程

4.3.6 解释器风格

- 应用实例： java class文件在JVM下通过解释器运行

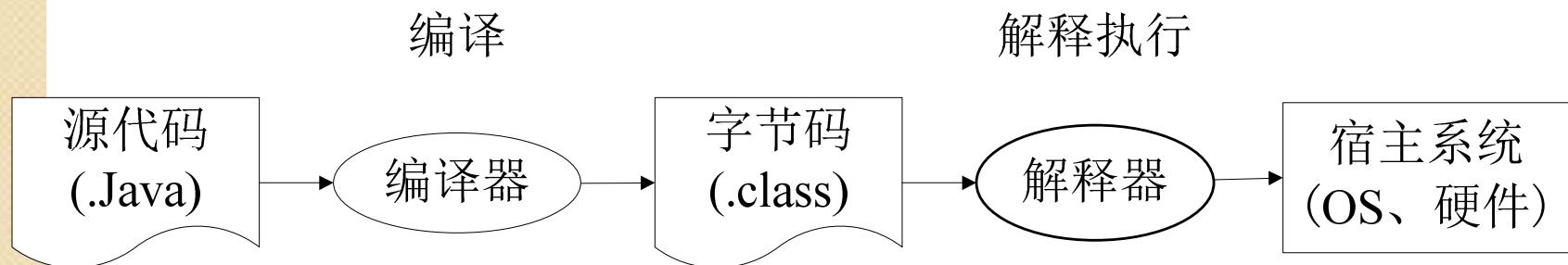


图4-17通过解释器执行.class文件

4.3.6 解释器风格

- 应用实例java class文件在JVM下通过JIT (just-in-time) 编译器运行

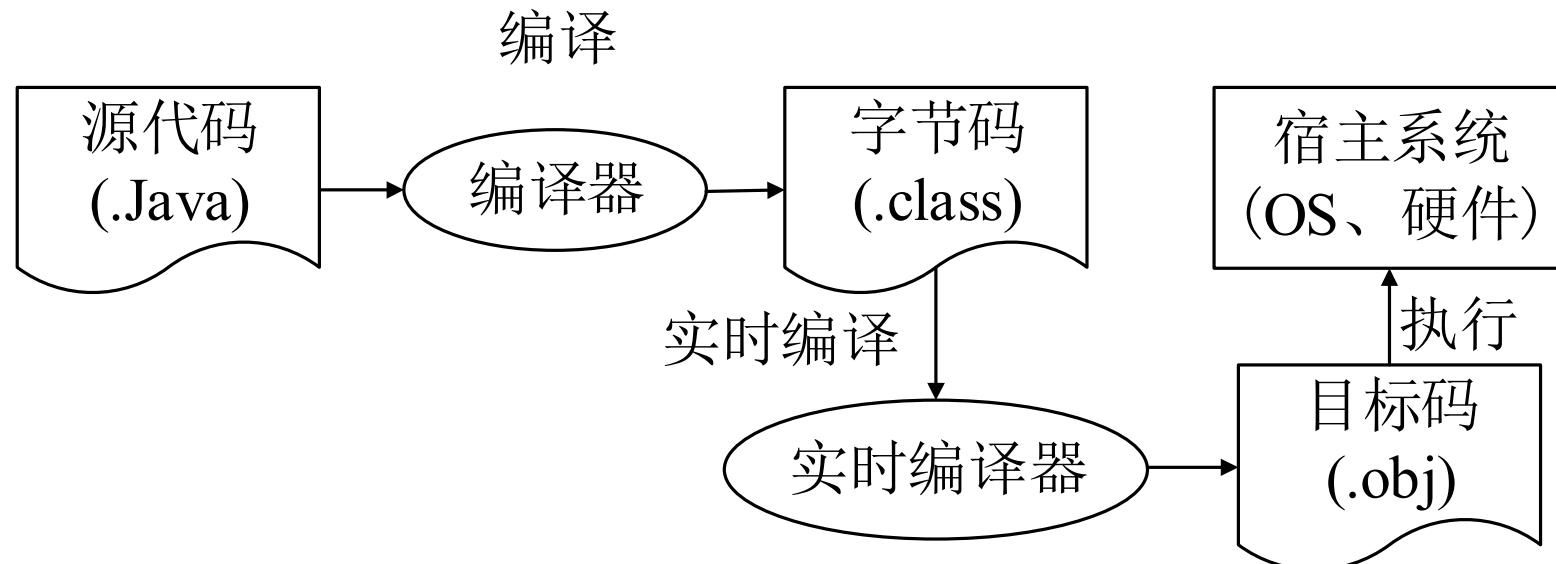


图4-18通过实时编译方式执行.class文件



4.3.7 基于规则的系统风格

- 现实里的业务需求经常频繁的发生变化，不断修改代码效率低、成本高。最好的办法是把频繁变化的业务逻辑抽取出来，形成独立的规则库。
- 这些规则可独立于软件系统而存在，可被随时的更新。
- 系统运行时，读取规则库，依据当前运行状态，从规则库中选择与之匹配的规则，对规则进行解释，根据结果控制系统运行的流程。



4.3.7 基于规则的系统风格

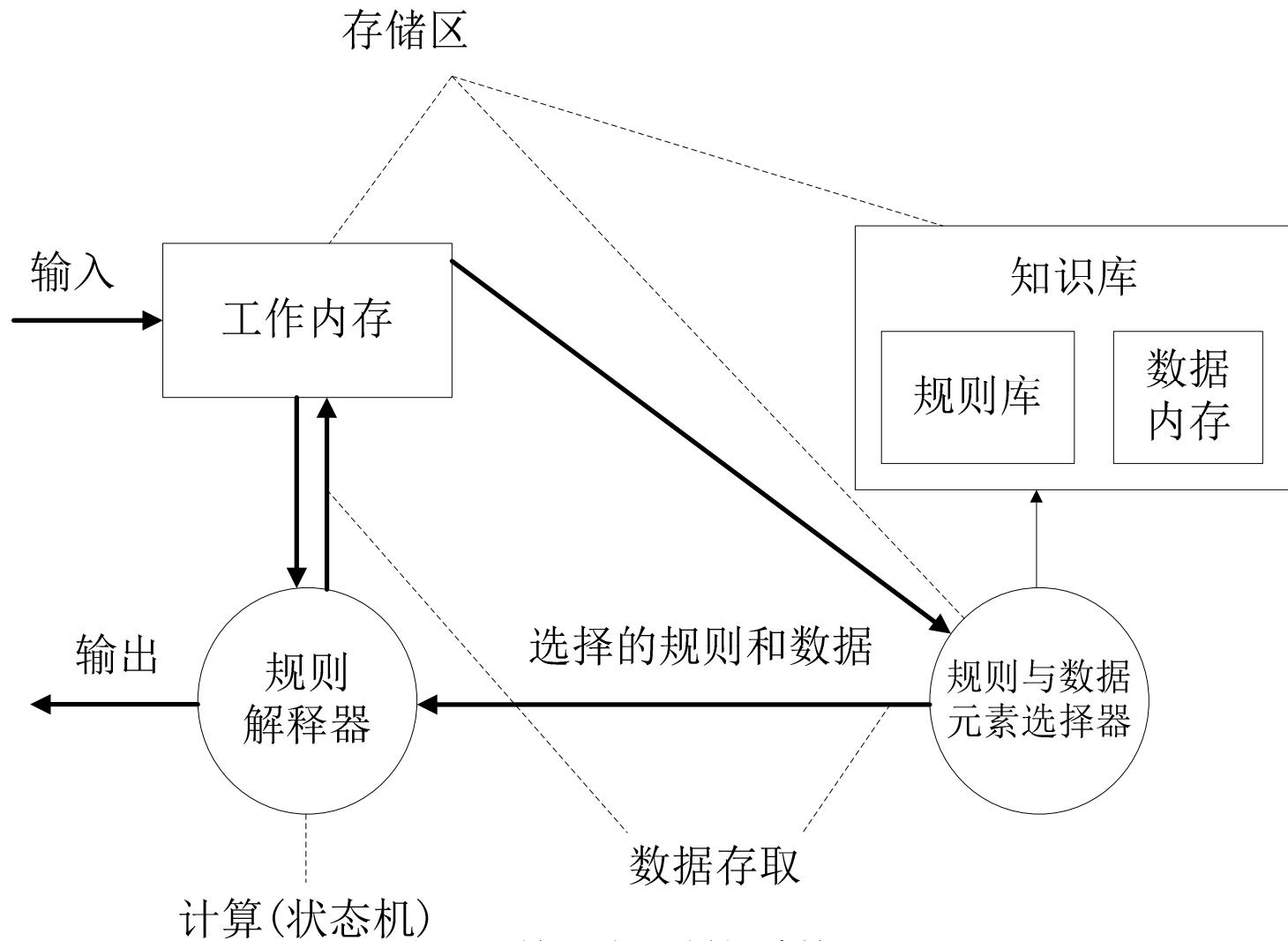
- 基于规则系统 (Rule-based System)
 - 一个使用模式匹配搜索来寻找规则并在正确的时候应用正确的逻辑知识的虚拟机。
 - 基于规则系统提供了一种基于专家系统解决问题的手段，将知识表示为“条件-行为”的规则，当满足条件时触发相应的行为，而不是将这些规则直接写在程序源代码中。

4.3.7 基于规则的系统风格

- 基于规则的系统风格的基本组件与解释器风格的组件相似

基于规则的系统	解释器风格
知识库	待解释的程序（伪码）
规则解释器	解释器引擎
规则与数据元素选择	解释器引擎内部的控制状态
工作内存	程序当前的运行状态

4.3.7 基于规则的系统风格



4.3.7 基于规则的系统风格

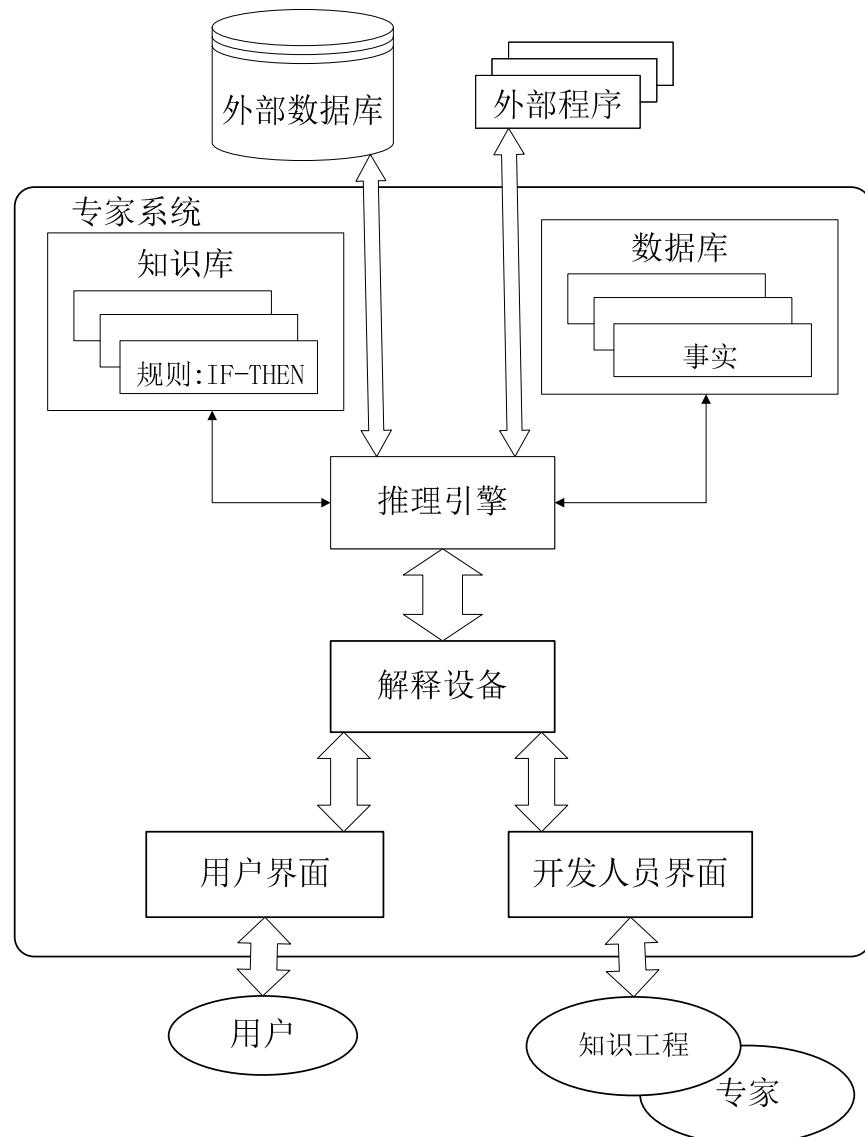
- 基于规则的系统和解释器风格的系统比较

	基于规则的系统	解释器风格的系统
相同点	基于规则的系统本质上是与解释器风格一致的，都是通过“解释器”(“规则引擎”), 在两个不同的抽象层次之间建立起一种虚拟的环境。	
不同点	在自然语言/XML的规则和高级语言的程序源代码之间建立虚拟机环境	在高级语言程序源代码和OS/硬件平台之间建立虚拟机环境

- 基于规则的系统的优缺点与解释器风格的系统类似。

4.3.7 基于规则的系统风格

- 应用实例
 - 基于规则的专家系统



4.3.8 仓库风格

- 基本思想

- 仓库是存储和维护数据的中心场所。
- 仓库式风格的两种组件：中央数据结构组件、相对独立的组件集合

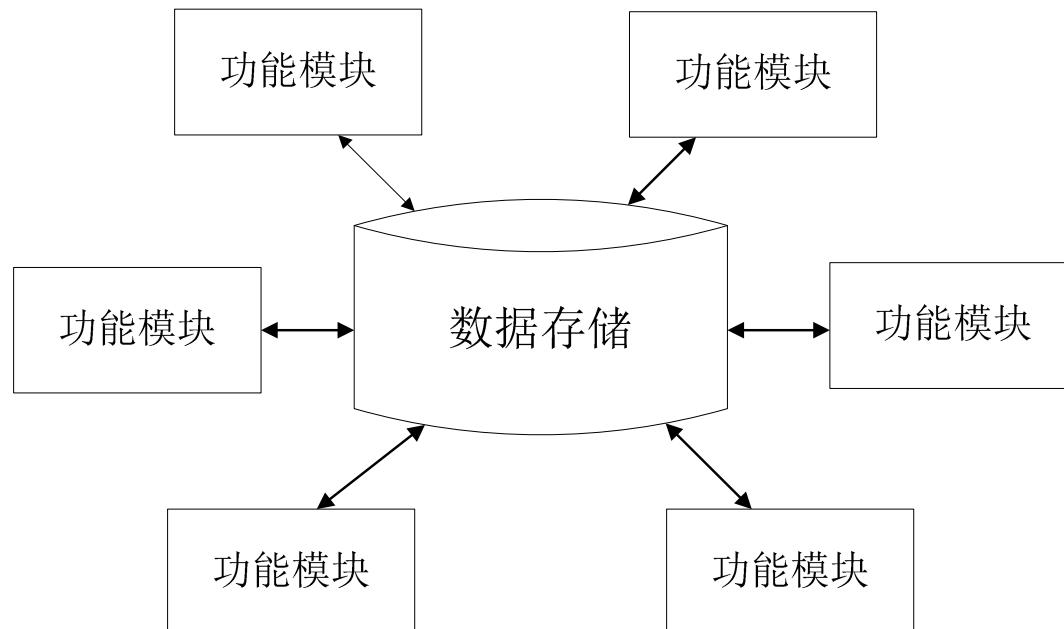


图4.21仓库风格

4.3.8 仓库风格

- 优点
 - 便于模块间的数据共享，方便模块的添加、更新和删除，避免了知识源的不必要的重复存储等。
- 缺点
 - 对于各个模块，需要一定的同步/加锁机制保证数据结构的完整性和一致性等。

4.3.8 仓库风格

- 应用实例：现代编译器

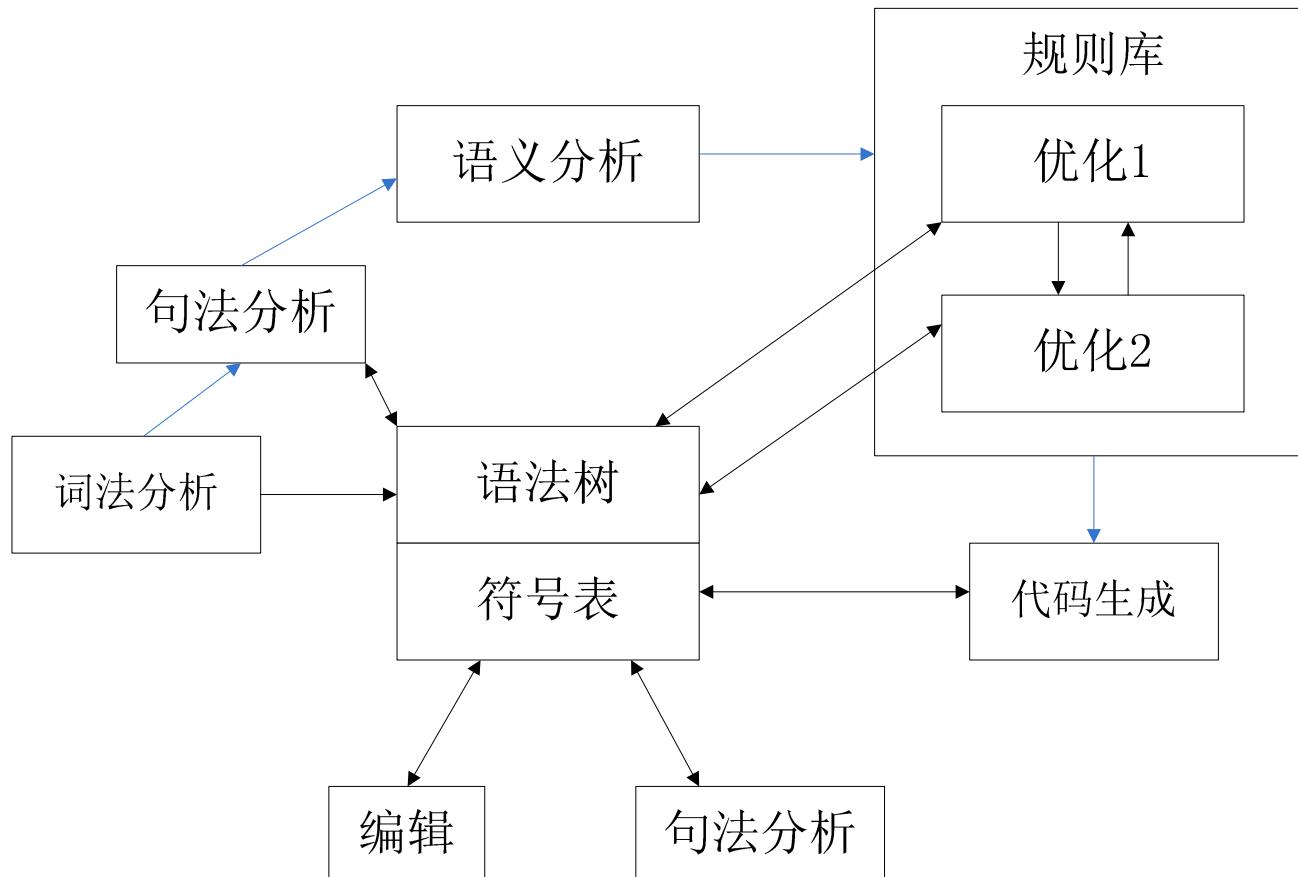


图4.22现代的规范编译器结构

4.3.8 仓库风格

- 应用实例：基于仓库风格的软件研发环境Eclipse

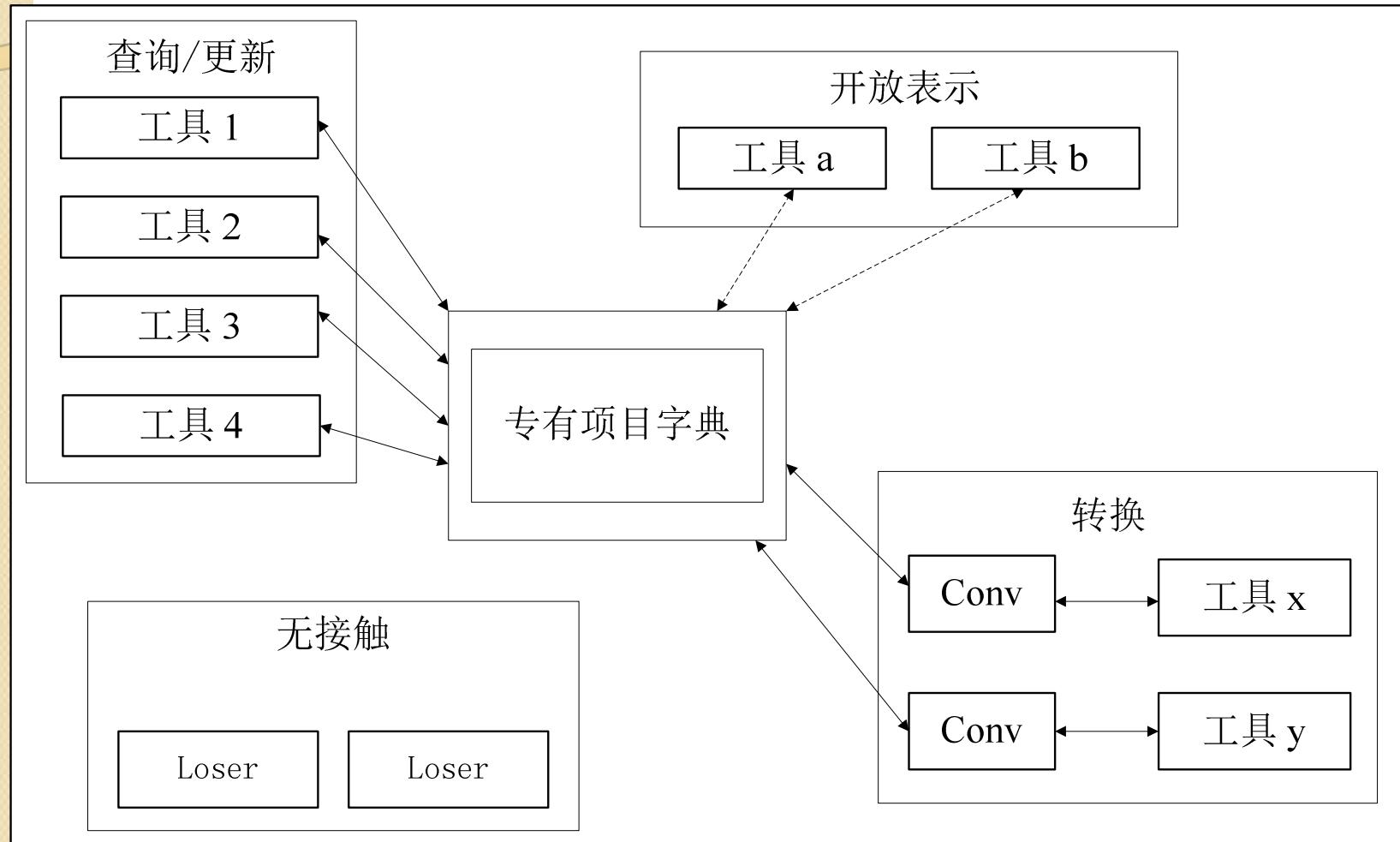


图4-23基于仓库风格的Eclipse软件研发环境



4.3.9 黑板系统风格

- 黑板系统（Blackboard System）是传统上被用于信号处理方面进行复杂解释的应用程序，以及松散耦合的组件访问共享数据的应用程序。
- 黑板架构实现的基本出发点是已经存在一个对公共数据结构进行协同操作的独立程序集合。

4.3.9 黑板系统风格

- 组成部分
 - (1) 知识源 (2) 黑板数据结构 (3) 控制器

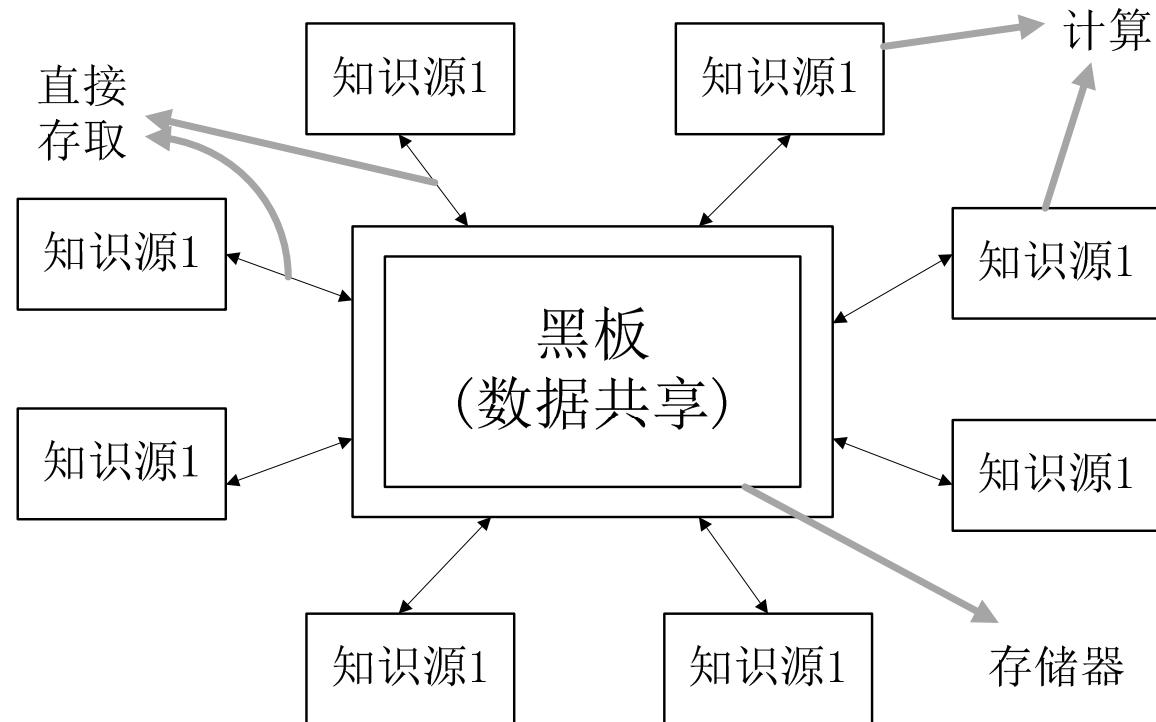


图4.24黑板架构风格



4.3.9 黑板系统风格

- 优点
 - (1) 便于多客户共享大量数据，他们不关心数据何时有的、谁提供的、怎样提供的。
 - (2) 既便于添加新的作为知识源代理的应用程序，也便于扩展共享的黑板数据结构。
 - (3) 知识源可重用。
 - (4) 支持容错性和健壮性。

4.3.9 黑板系统风格

- 缺点
 - (1) 不同的知识源代理对于共享数据结构要达成一致，而且，这也造成对黑板数据结构的修改较为困难——要考虑到各个代理的调用。
 - (2) 需要一定的同步/加锁机制保证数据结构的完整性和一致性，增大了系统复杂度。

4.3.9 黑板系统风格

- 应用实例
 - HEARSAY-II

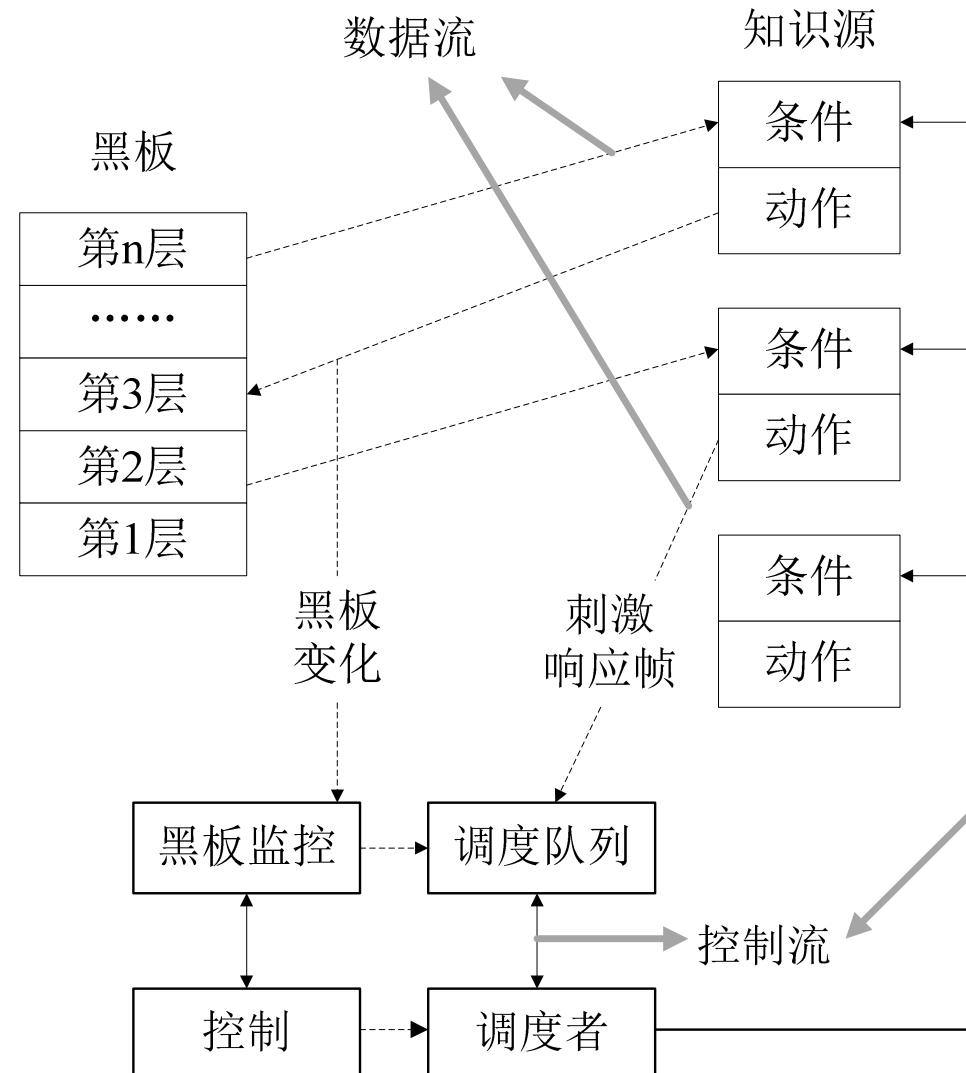


图4.25 HEARSAY-II架构

4.3.10 C2风格

- C2结构是1995年由California大学Irvine分校的Richard. N. Taylor等人提出
- C2是一种基于组件和消息的架构风格，适用于GUI软件开发，构建灵活和可扩展的应用系统。
- C2风格的主要思想来源于Chiron-1用户界面系统，因此又被命名为Chiron-2，简称C2。
- C2架构风格可以概括为：通过连接件绑定在一起的按照一组规则运作的并行组件网络。



4.3.10 C2风格

- C2风格的系统组织规则：
 - (1) 系统中的组件和连接件都有一个顶部和一个底部；
 - (2) 组件的顶部应连接到某连接件的底部，组件的底部则应连接到某连接件的顶部，不允许组件之间的直接连接
 - (3) 一个连接件可以和任意数目的其他组件和连接件连接；
 - (4) 当两个连接件进行直接连接时，必须由其中一个的底部到另一个的顶部。

4.3.10 C2风格

- 其结构如图所示：

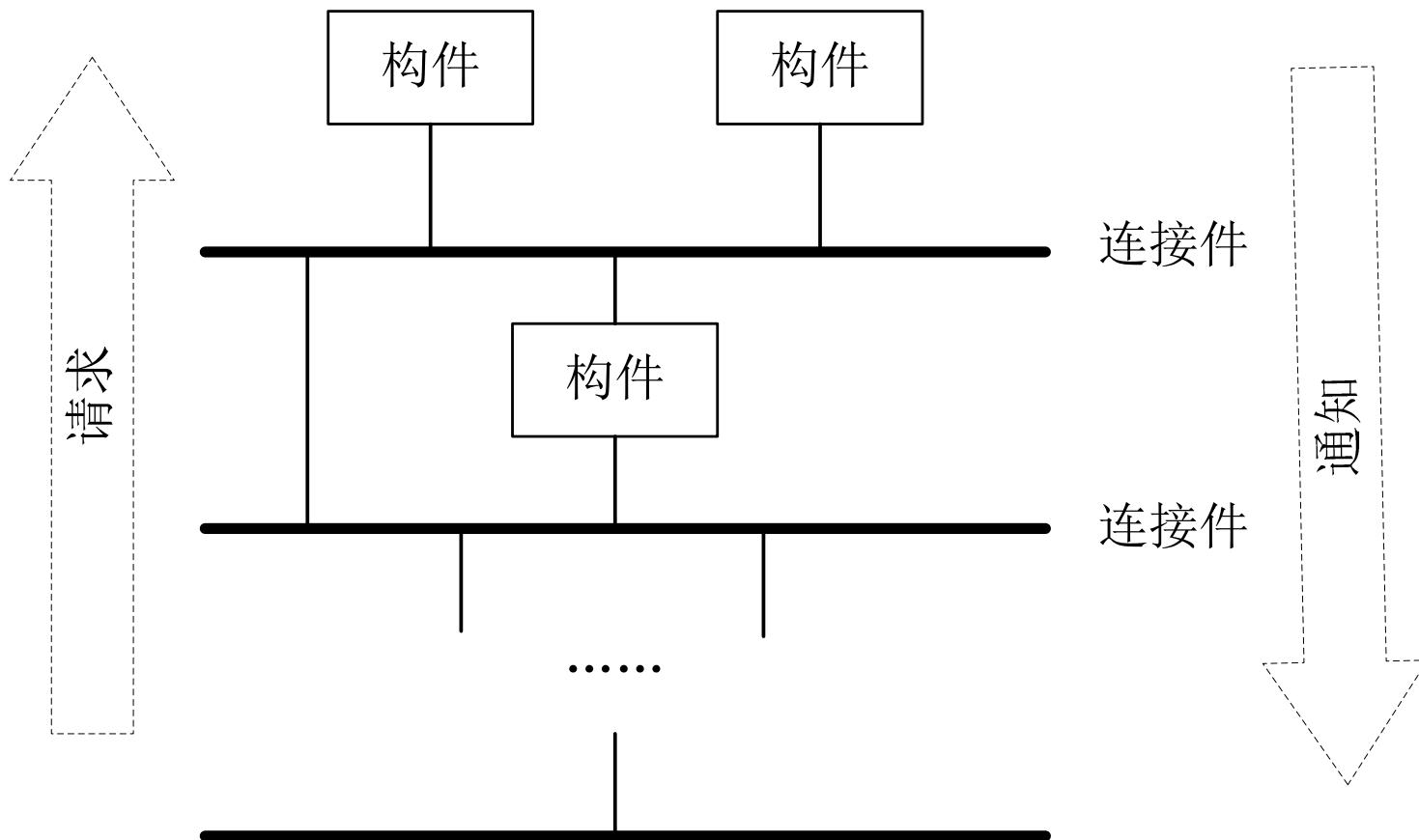


图4.26 C2风格

4.3.10 C2风格

- C2架构的内部，通信和处理是分开完成的。

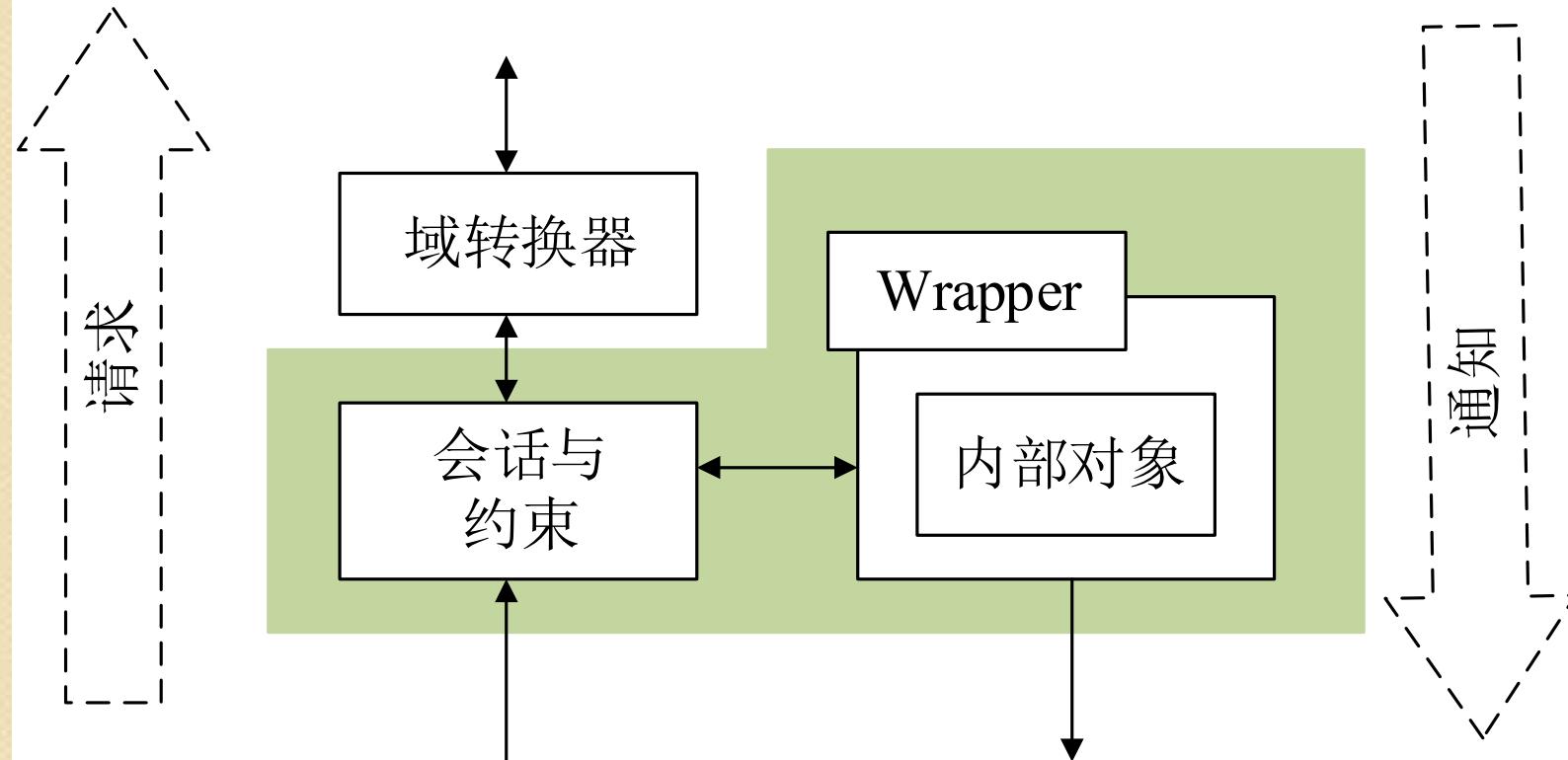


图4-27 C2组件的内部结构

4.3.10 C2风格

- 优点：
 - (1) 可使用任何编程语言开发组件，组件重用和替换易实现；
 - (2) 由于组件之间相对独立，依赖较小，因而该风格具有一定扩展能力，可支持不同粒度的组件；
 - (3) 组件不需共享地址空间；
 - (4) 可实现多个用户和多个系统之间的交互；
 - (5) 可使用多个工具集和多种媒体类型，动态更新系统框架结构。



4.3.10 C2风格

- 缺点
 - 不太适合大规模流式风格系统，以及对数据库使用比较频繁的使用。

4.3.10 C2风格

- 应用实例
 - KLAX游戏

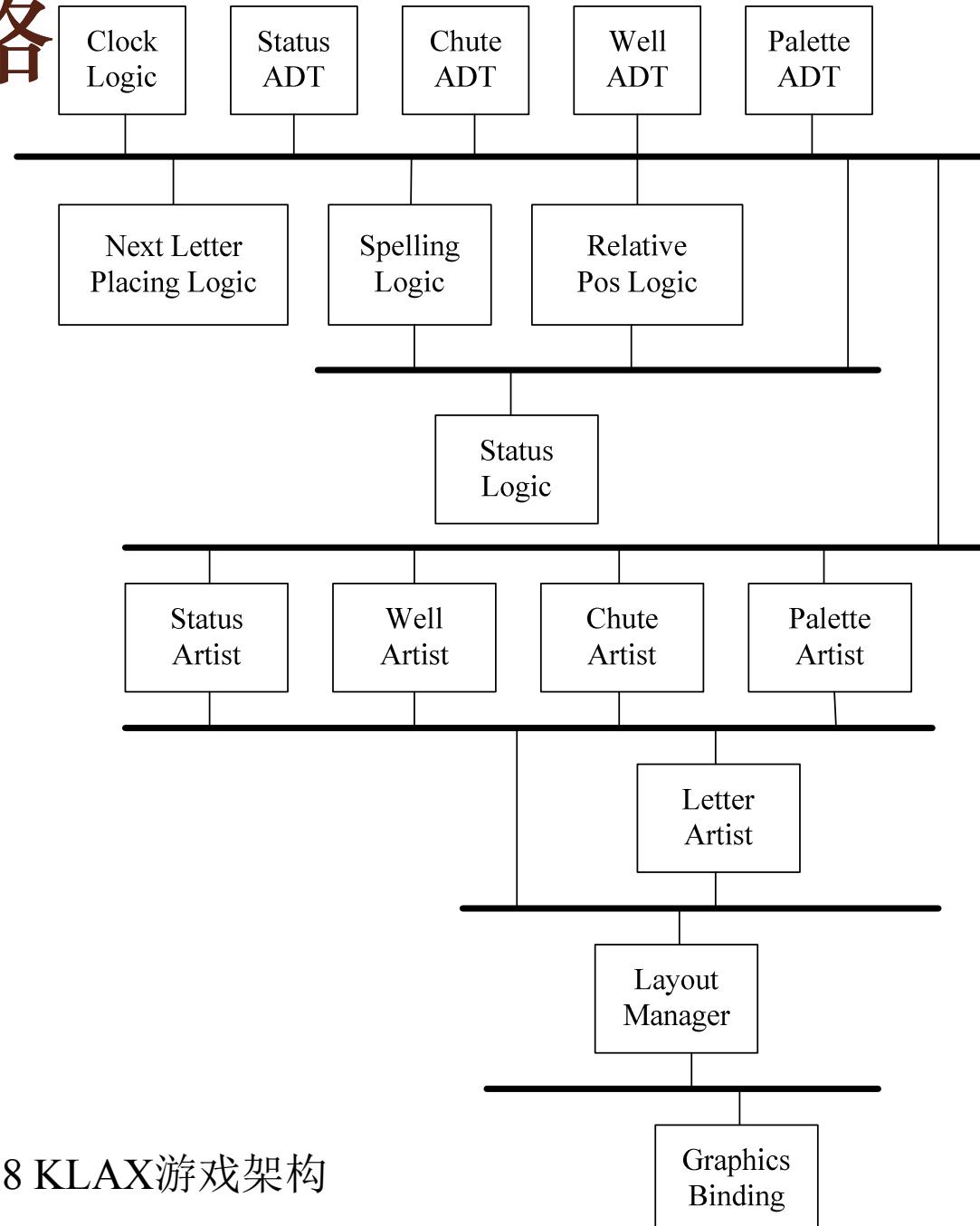


图4.28 KLAX游戏架构

4.3.11 客户机/服务器风格

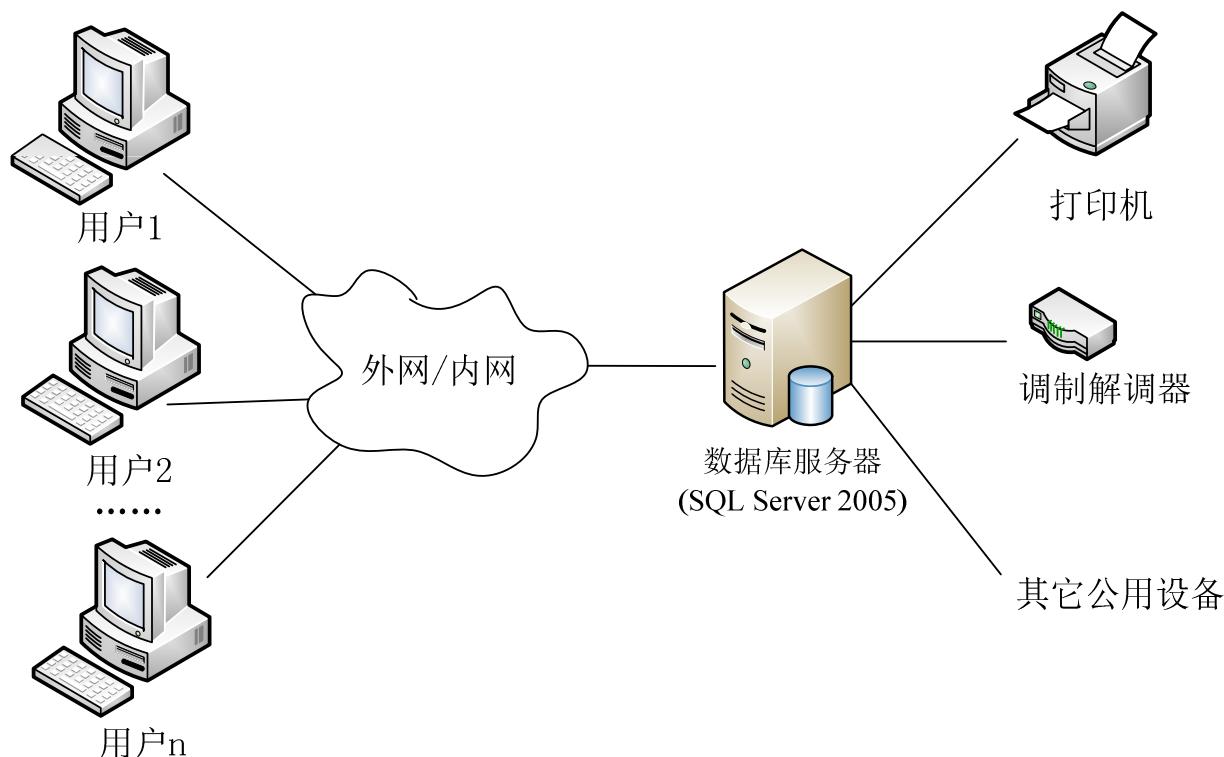
- 客户机/服务器 (Client/Server)是20世纪90年代开始成熟的一项技术，主要针对资源不对等问题而提出的一种共享策略。
- 客户机和服务器是两个相互独立的逻辑系统，为了完成特定的任务而形成一种协作关系。
 - 客户机(前端， front-end): 业务逻辑、与服务器通讯的接口；
 - 服务器(后端： back-end): 与客户机通讯的接口、业务逻辑、数据管理。

4.3.11 客户机/服务器风格

- 一般的，客户机为完成特定的工作向服务器发出请求；服务器处理客户机的请求并返回结果。
- 客户机程序和服务器程序配置在同一台计算机上时：采用消息、共享存储区和信号量等方法实现通信连接。
- 客户机程序和服务器程序配置在分布式环境中时：通过远程调用（Remote Produce Call, RPC）协议来进行通信。

4.3.11 客户机/服务器风格

- 基本思想
 - 两层C/S架构



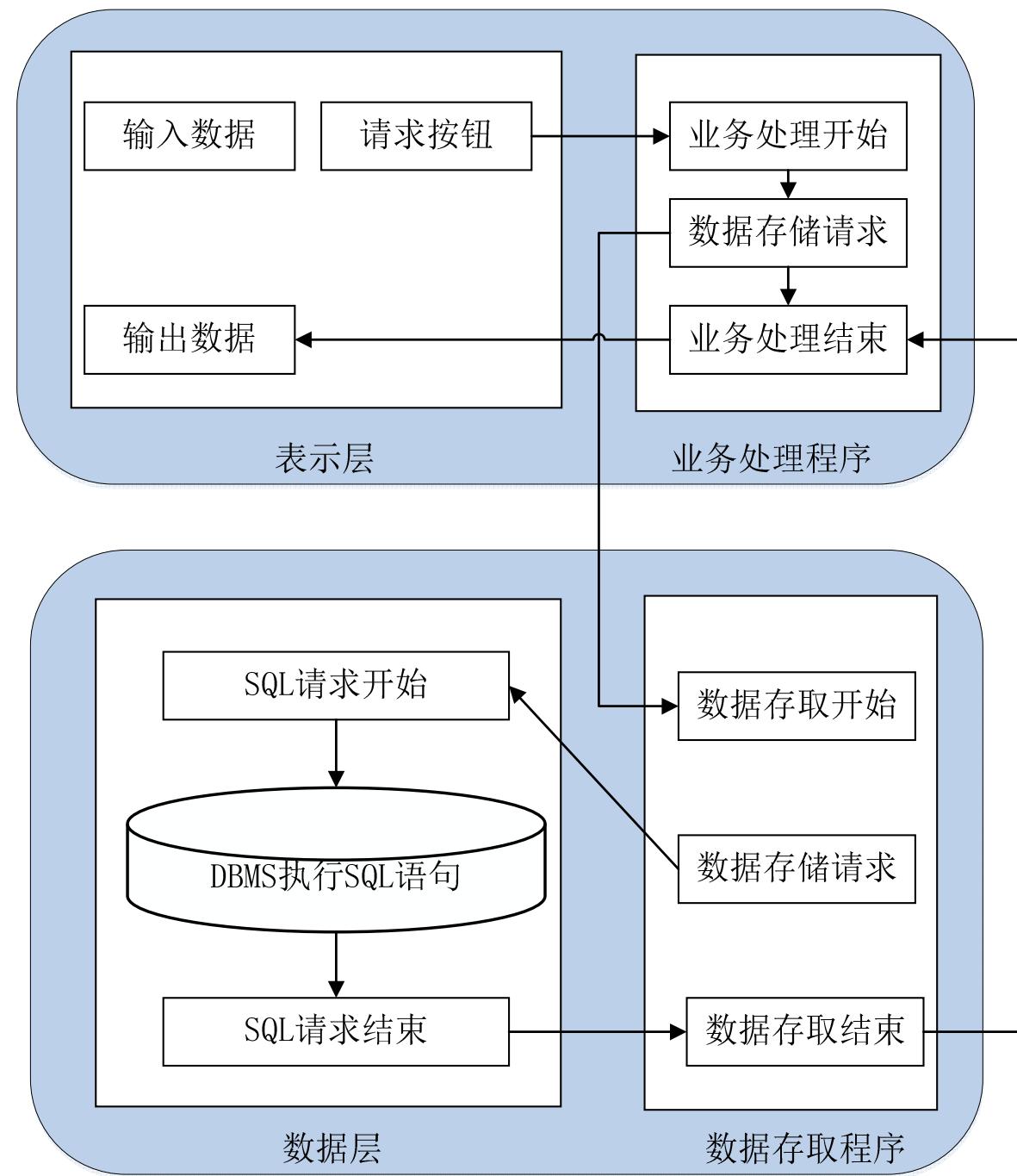


图4-30两层C/S架构的处理流程

4.3.11 客户机/服务器风格

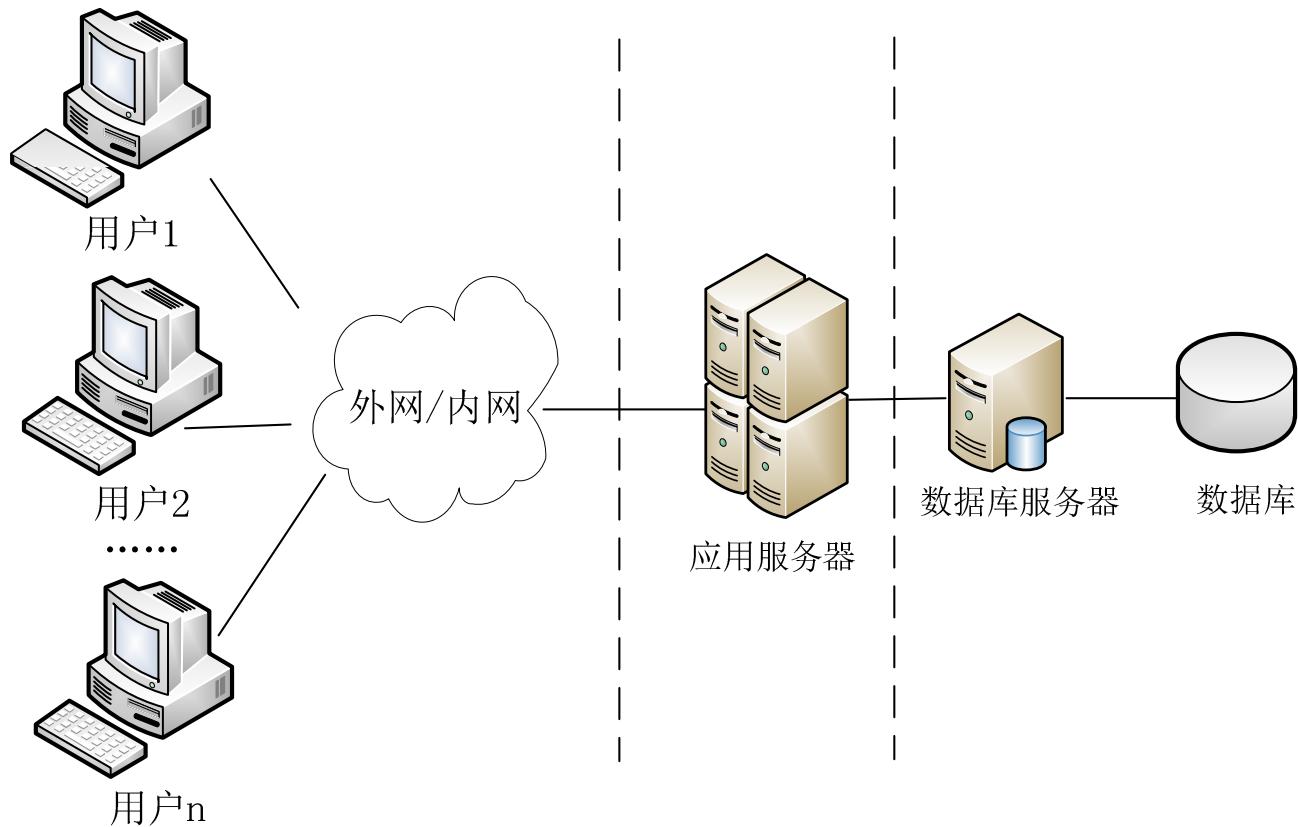
- 两层C/S架构优点：
 - (1) 客户机组件和服务器组件分别运行在不同的计算机上，有利于分布式数据的组织和处理。
 - (2) 组件之间的位置是相互透明的
 - (3) 客户机程序和服务器程序可运行在不同的操作系统上，便于实现异构环境和多种不同开发技术的融合。
 - (4) 软件环境和硬件环境的配置具有极大的灵活性，易于系统功能的扩展。
 - (5) 将大规模的业务逻辑分布到多个通过网络连接的低成本的计算机上，降低了系统的整体开销。

4.3.11 客户机/服务器风格

- 两层C/S架构缺点：
 - (1) 开发成本较高（客户机的软硬件要求高）。
 - (2) 客户机程序的设计复杂度大，客户机负荷重。
 - (3) 信息内容和形式单一。
 - (4) C/S架构升级需要开发人员到现场更新客户机程序，对运行环境进行重新配置，增加了维护费用。
 - (5) 两层C/S结构采用了单一的服务器，同时以局域网为中心，难以扩展到Intranet和Internet。
 - (6) 数据安全性不高。

4.3.11 客户机/服务器风格

- 基本思想
 - 三层C/S架构



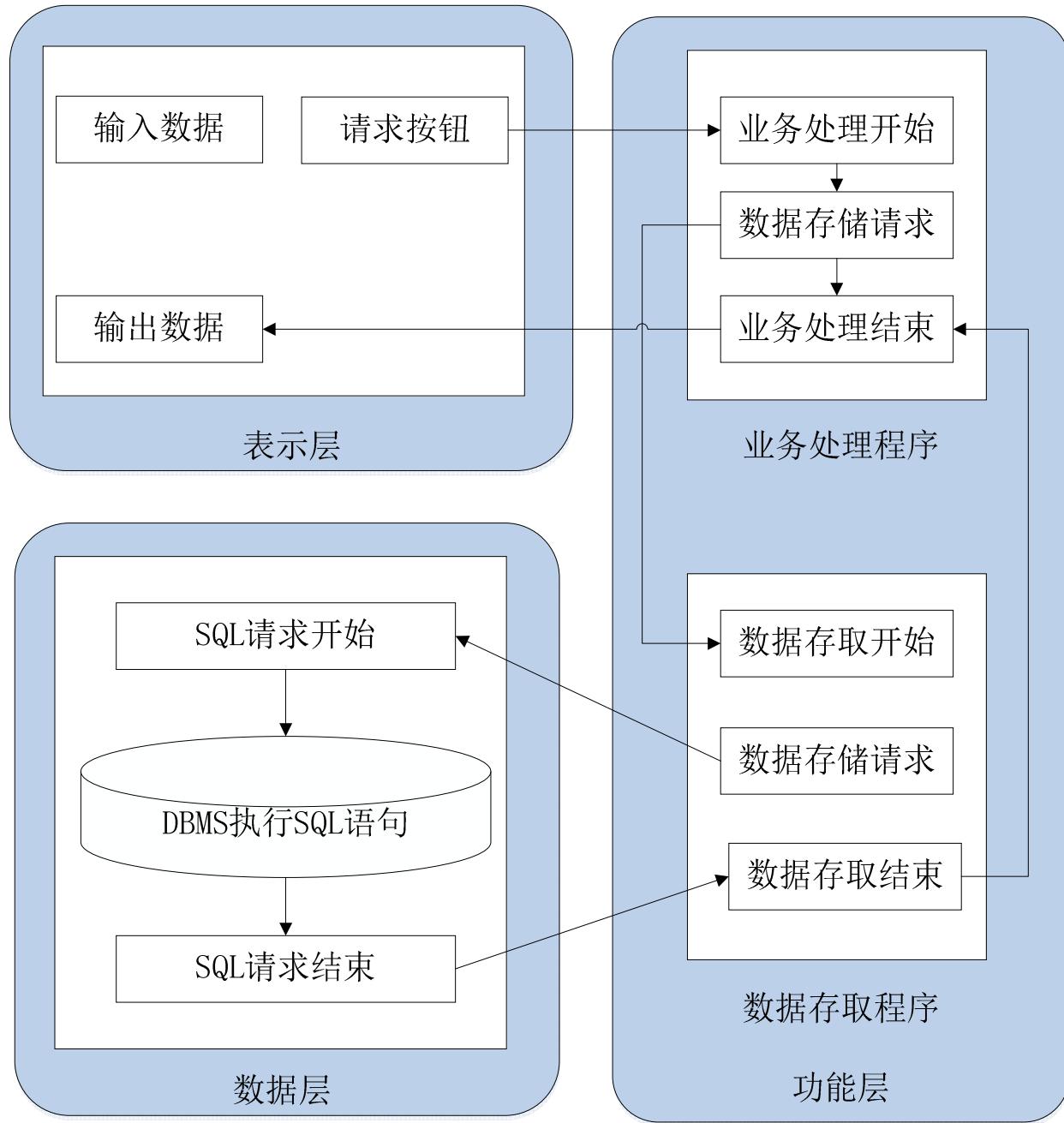
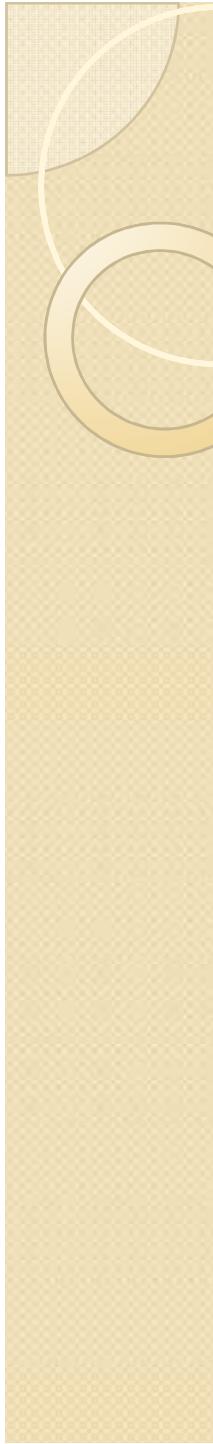
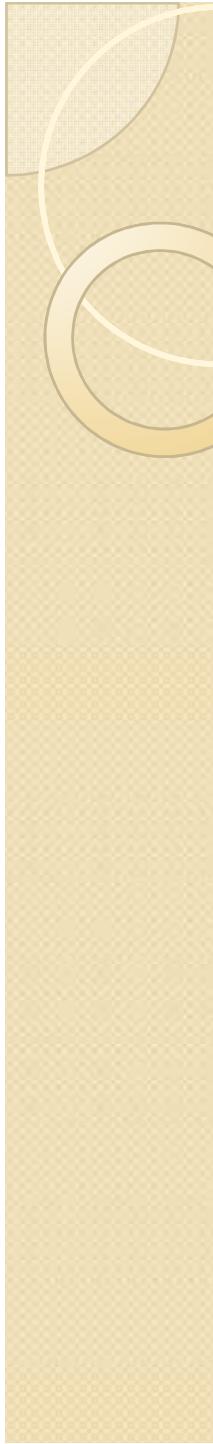


图4-32三层C/S架构的处理流程



4.3.11 客户机/服务器风格

- 三层C/S架构相比于两层C/S架构的优点：
 - (1) 合理地划分三层结构的功能，可以使系统的逻辑结构更加清晰，提高软件的可维护性和可扩充性。
 - (2) 在实现三层C/S架构时，可以更有效地选择运行平台和硬件环境，从而使每一层都具有清晰的逻辑结构、良好的负荷处理能力和较好的开放性。
 - (3) 在C/S架构中，可以分别选择合适的编程语言并行开发。
 - (4) 系统具有较高的安全性。

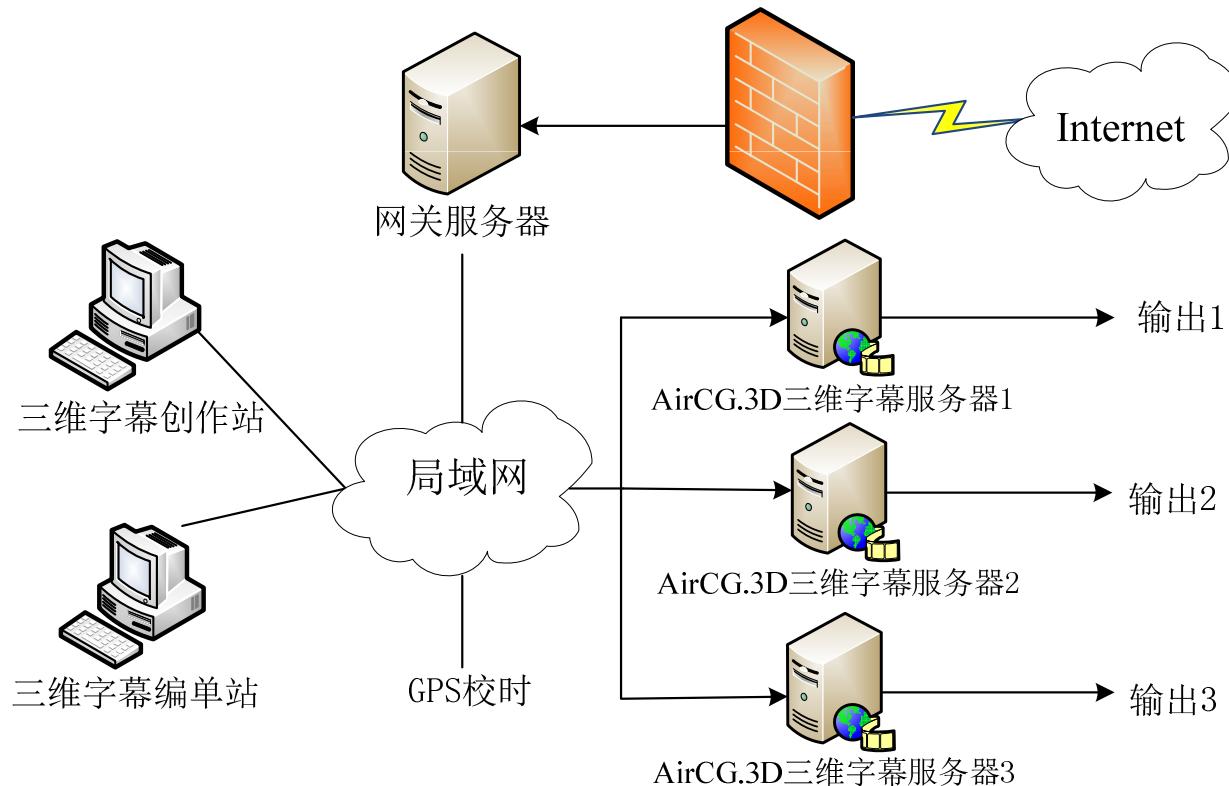


4.3.11 客户机/服务器风格

- 在使用三层C/S架构时需注意以下两个问题：
 - (1) 如果各层之间的通信效率不高，即使每一层的硬件配置都很高，系统的整体性能也不会太高。
 - (2) 必须慎重考虑三层之间的通信方法、通信频率和传输数据量，这和提高各层的独立性一样也是实现三层C/S架构的关键性问题。

4.3.11 客户机/服务器风格

- 应用实例
 - AirCG.3D三维网络字幕系统的结构



4.3.12 浏览器/服务器风格

- 浏览器/服务器风格是三层C/S风格的一种实现方式。
- 主要包括浏览器，Web服务器和数据库服务器。

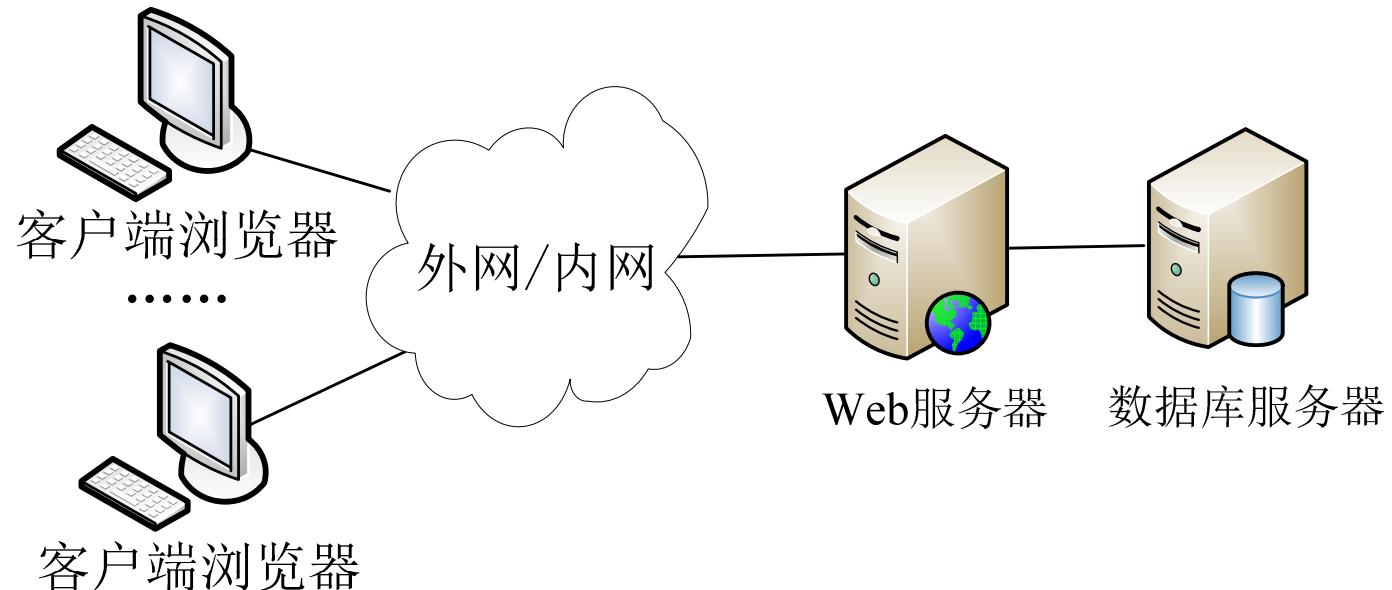


图4.34 B/S架构

4.3.12 浏览器/服务器风格

- 与三层C/S结构的解决方案相比，B/S架构在客户机上采用了WWW浏览器，将Web服务器作为应用服务器。
- B/S架构核心是Web服务器，数据请求、网页生成、数据库访问和应用程序执行全部由Web服务器来完成。
- 在B/S架构中，系统安装、修改和维护全在服务器端解决，客户端无任何业务逻辑。

4.3.12 浏览器/服务器风格

- 优点

- (1) 客户端只需要安装浏览器，操作简单，能够发布动态信息和静态信息。
- (2) 运用HTTP标准协议和统一客户端软件，能够实现跨平台通信。
- (3) 开发成本比较低，只需要维护Web服务器程序和中心数据库。客户端升级可以通过升级浏览器来实现。

4.3.12 浏览器/服务器风格

- 缺点

- (1) 个性化程度比较低，所有客户端程序的功能都是一样的。
- (2) 客户端数据处理能力比较差，加重了Web服务器的工作负担，影响系统的整体性能。
- (3) 在B/S架构中，数据提交一般以页面为单位，动态交互性不强，不利于在线事物处理(Online Transaction Processing, OLTP)。
- (4) B/S架构的可扩展性比较差，系统安全性难以保障。
- (5) B/S架构的应用系统查询中心数据库，其速度要远低于C/S架构。

4.3.12 浏览器/服务器风格

- 应用实例

- PetShop

PetShop架构分为三层：

(1) 数据访问层：其功能主要是负责数据库的访问。

(2) 业务逻辑层：是整个系统的核心，它与这个系统的业务（领域）有关。PetShop的业务逻辑层的相关设计，均和网上宠物店特有的逻辑相关。

(3) 表示层：是系统的UI部分，负责使用者与整个系统的交互。在这一层中，理想的状态是不应包括系统的业务逻辑。表示层中的逻辑代码，仅与界面元素有关。



4.3.13 平台/插件风格

- 插件（Plug-in）是一种遵循统一的预定接口规约编写出来的程序，应用程序在运行时通过接口规约对插件进行调用，以扩展应用程序的功能。
- 插件的本质在于不修改程序主体（或者程序运行平台）的情况下对软件功能进行扩展与加强。

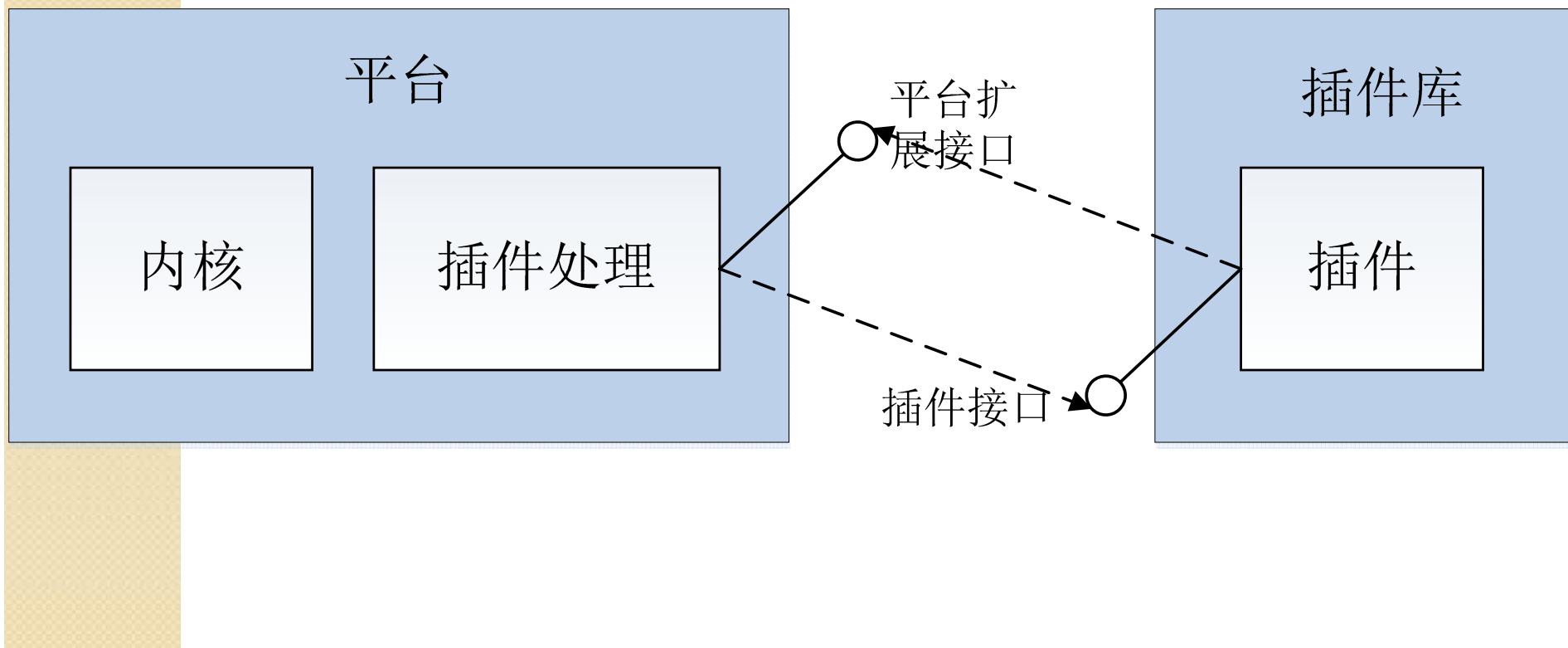
4.3.13 平台/插件风格

- “平台/插件”软件结构将待开发的目标软件分为两部分：
 - 程序的主体或主框架，可定义为平台
 - 功能扩展或补充模块，可定义为插件
- 平台所完成的功能应为一个软件系统的核心和基础,可以把平台基本功能分为两个部分：
 - 内核功能：是整个软件的重要功能，一个软件的大部分功能因由内核功能完成。
 - 插件处理功能：用于扩展平台和管理插件，为插件操纵平台和与插件通信提供标准平台扩展接口。
 - 如：Eclipse IDE是Eclipse的运行主体平台，编辑c/c++程序可以应用CDT插件；使用SVN可以安装SVN Repository Exploring。

4.3.13 平台/插件风格

- 实现“平台/插件”(Platform / Plug-in) 结构软件需要定义两个标准接口
 - 平台扩展接口：完全由平台实现，插件只是调用和使用
 - 插件接口：完全由插件实现，平台也只是调用和使用。
 - 平台扩展接口实现插件向平台方向的单向通信，插件通过平台扩展接口可获取主框架的各种资源和数据，可包括各种系统句柄，程序内部数据以及内存分配等。
 - 插件接口为平台向插件方向的单向通信，平台通过插件接口调用插件所实现的功能，读取插件处理数据等。

4.3.13 平台/插件风格



4.3.13 平台/插件风格

- 优点：
 - (1) 降低系统各模块之间的互依赖性
 - (2) 系统模块独立开发、部署、维护
 - (3) 根据需求动态的组装、分离系统
- 缺点
 - 插件是别人开发的可以用到某主程序中的，只服务于该主程序，可重用性差。

4.3.13 平台/插件风格

- 应用实例
 - Eclipse开发平台

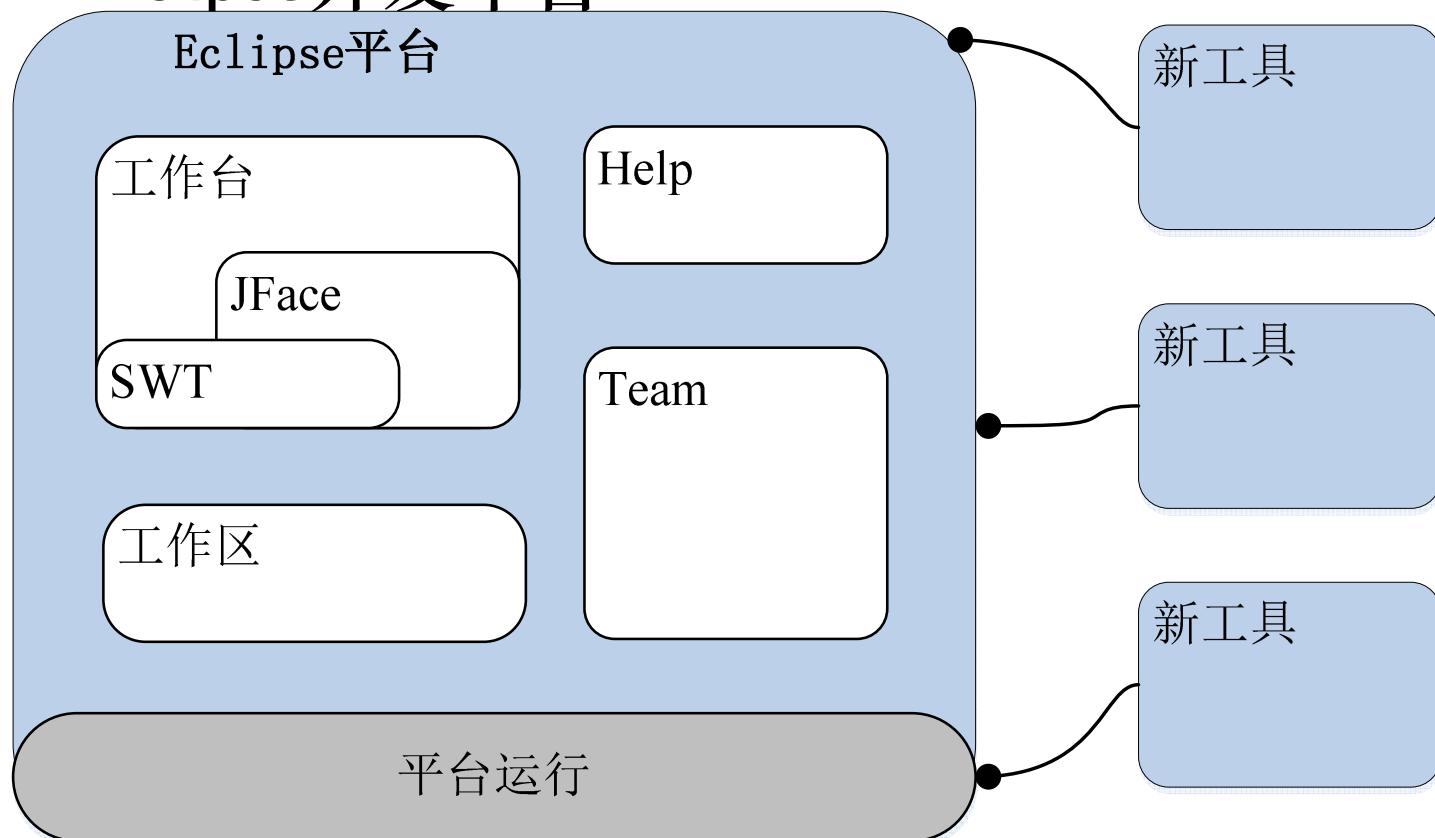


图4.36 Eclipse的插件机制

4.3.14 面向Agent风格

- 面向Agent风格 (Agent-oriented) 的基本思想是认为事物的属性，特别是动态特性在很大程度上受到与其密切相关的人和环境的影响，将影响事物的主观与客观特征相结合抽象为系统中的Agent，作为系统的基本构成单位，通过Agent之间的合作实现系统的整体目标。

4.3.14 面向Agent风格

- Agent：一个能够根据它对其环境的感知，主动采取决策和行为的软件实体。
- Agent组件：对系统处理的高度抽象，具有高度灵活和高度智能特色的软件实体。
- Agent连接件：对复合型组件的连接，该连接能够提供通信、协调、转换、接通等服务。



4.3.14 面向Agent风格

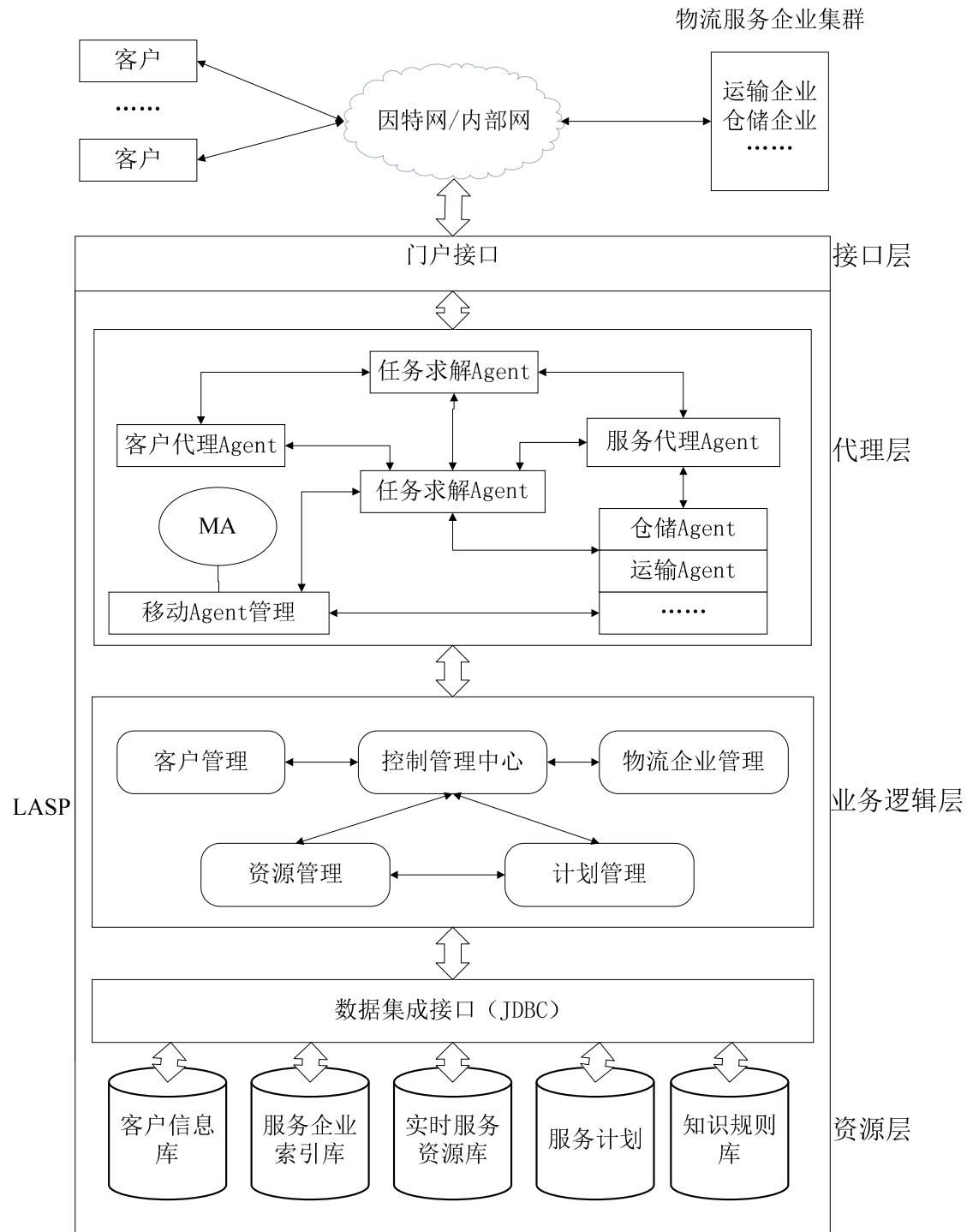
- Agent组件有别于以往任何系统的组件类型，其所具有的自主性、智能性、交互性等特性是传统架构对象所不具备的。
- 多Agent系统中的连接件并非显式地将两个不同的组件联系起来。不同Agent之间的联系是根据运行时状态来决定的。



4.3.14 面向Agent风格

- 优点
 - 面向Agent的软件工程方法对于解决复杂问题是一种好的技术,特别是对于分布开放异构的软件环境。
- 缺点
 - 大多数结构中Agent自身缺乏社会性结构描述和与环境的交互。

图4-38LASP物流管理系统的架构





4.3.15 面向方面软件架构风格

- 基本思想
 - 一般认为AOP在传统软件架构基础上增加了方面组件(Aspect Component)这一新的构成单元，通过方面组件来封装系统的横切关注点。
 - 系统的有些特性和需求是横切于系统的每一个层面中，并融于系统的每一个组件中，这种特性称为系统的方面(Aspect)需求特性或关注点
 - 如系统中的时间要求、业务逻辑、性能、安全性和错误检测、QoS监测等。
 - 一个复杂的软件系统可以看作是由许多关注点的实现构成的。



4.3.15 面向方面软件架构风格

- 例如：
 - 企业应用系统中每一个业务流程的实现就是系统的一个功能或子系统，在每一个业务流程中横切着控制流、数据流和参与者操作等方面的关注，它们反映相互正交的业务逻辑：业务流程、业务活动和参与者。

4.3.15 面向方面软件架构风格

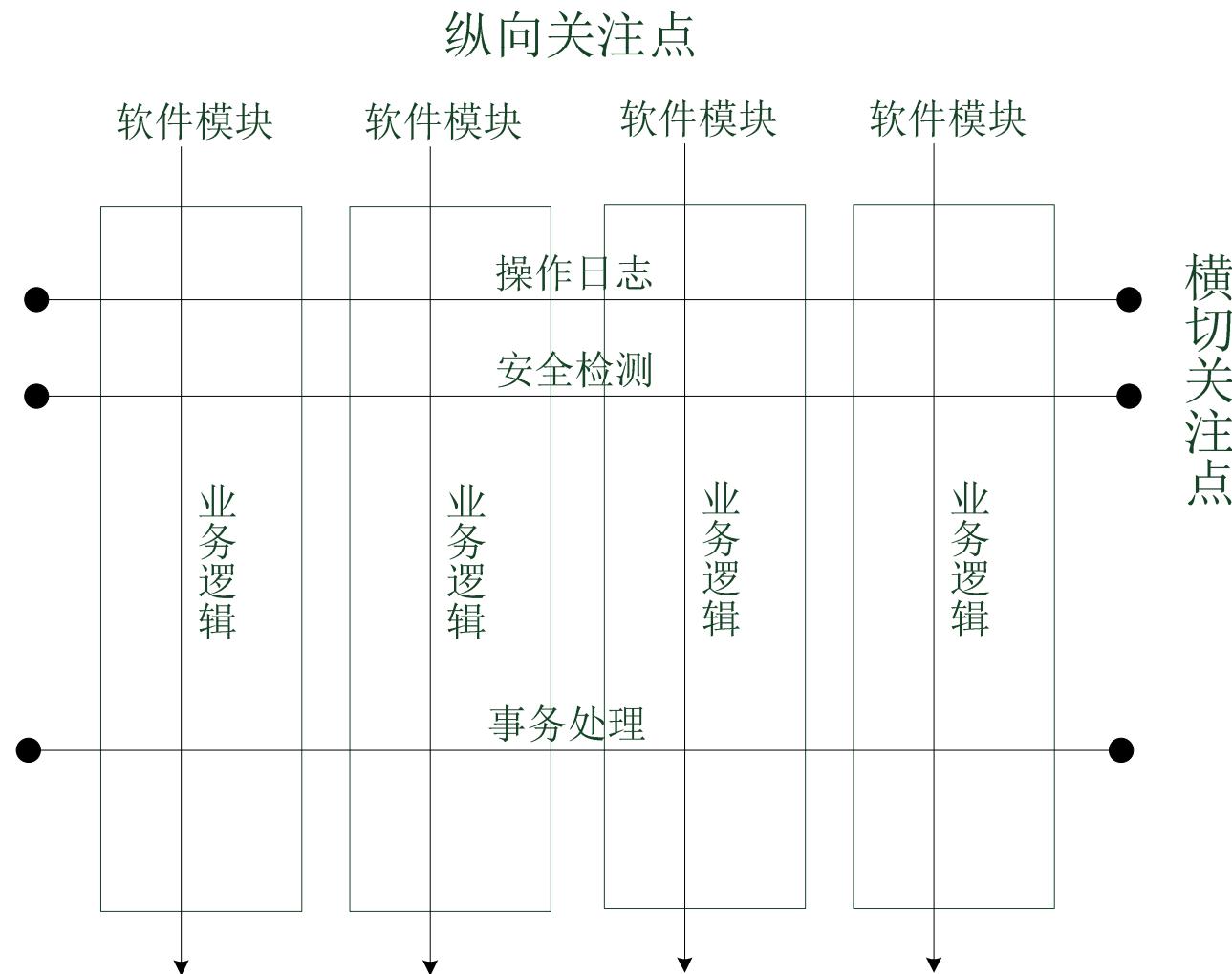


图4.39 AOP示意图



4.3.15 面向方面软件架构风格

- 优缺点分析

- 可以定义交叉的关系，并将这些关系应用于跨模块的、彼此不同的对象模型
- AOP 同时还可以让我们层次化功能性而不是嵌入功能性，从而使得代码有更好的可读性和易于维护。
- 它会和面向对象编程可以很好地合作，互补

4.3.15 面向方面软件架构风格

- 应用实例

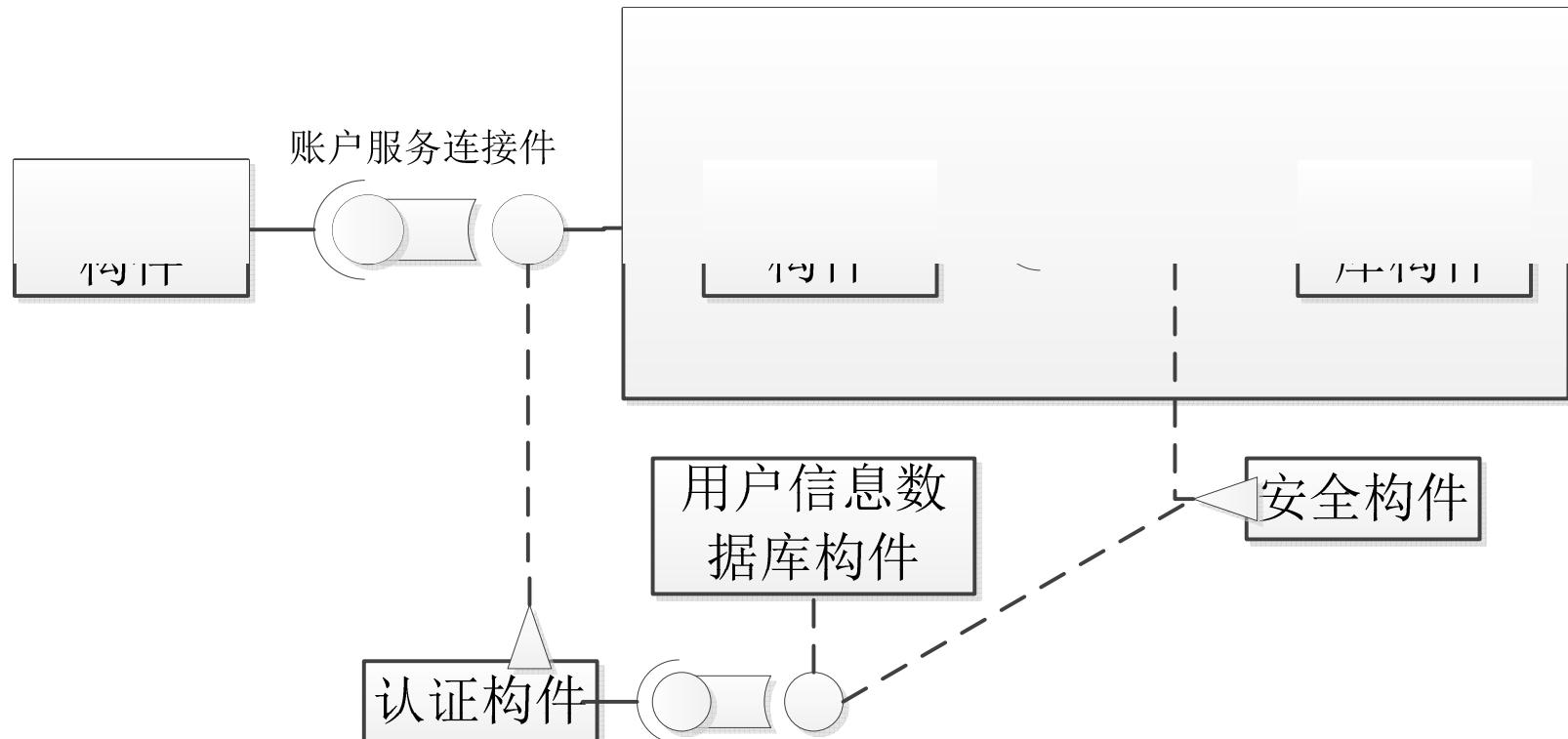


图4.40 网上支付模型



4.3.16 面向服务架构风格

- 1996年，Gartner Group 提出面向服务架构模型SOA
- SOA 是一个组件模型，它将应用程序的不同功能单元（服务）通过这些服务之间定义良好的接口和契约联系起来。
 - 服务(service)是一个粗粒度的、可发现的软件实体。
 - 接口是采用中立的方式进行定义的，应独立于实现服务的硬件平台、操作系统和编程语言，便于不同系统中的服务以一种统一和通用的方式进行交互。

4.3.16 面向服务架构风格

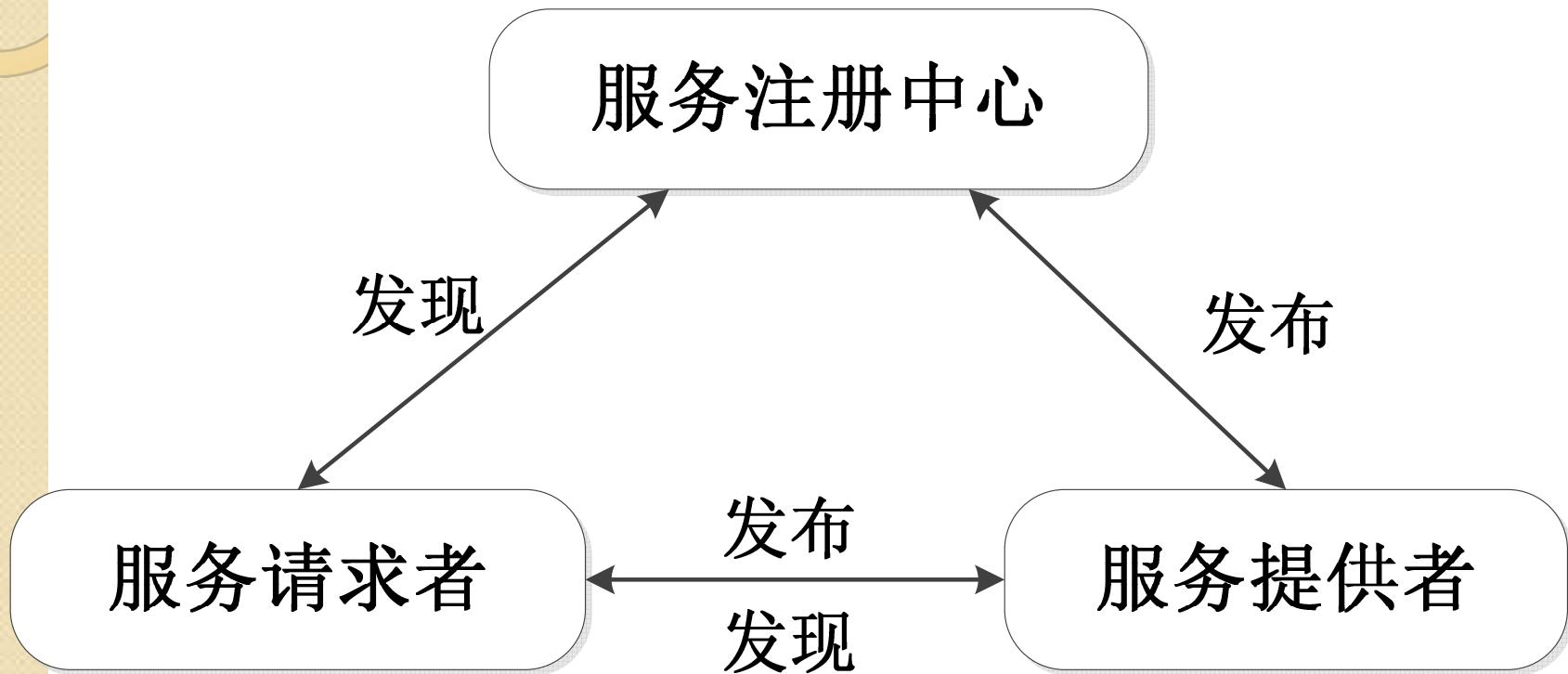


图4.41 SOA架构

4.3.16 面向服务架构风格

- 服务请求者：可以是服务或者第三方的用户，通过查询服务提供者在服务注册中心发布的服务接口的描述，通过服务接口描述来通过RPC或者SOAP进行绑定调用服务提供者所提供的业务或服务。
- 服务提供者：作为服务管理者和创建者，必须将服务描述的接口发布到服务注册中心才能被潜在的服务请求发现，能够为合适的服务请求者提供服务。
- 服务注册中心：相当于服务接口的管理中心，服务请求者能够通过查询服务注册中心的数据库来找到需要的服务调用方式和接口描述信息。

4.3.16 面向服务架构风格

- **发布：**为了便于服务请求者发现，服务提供者将对服务接口的描述信息发布到服务注册中心上。
- **发现：**服务请求者通过查询服务注册中心的数据库来找到需要的服务，服务注册中心能够通过服务的描述对服务进行分类，使服务提供者更快定位所需要的服务范围。
- **绑定和调用：**服务请求者在查询到所需要服务描述信息，根据这些信息服务请求者能够调用服务。

4.3.16 面向服务架构风格

- 面向服务软件架构风格在于具有基于标准、松散耦合、共享服务和粗粒度等优势，表现为易于集成现有系统、具有标准化的架构、提升开发效率、降低开发维护复杂度等。
- 通过采用SOA架构，在进行一次开发成本急剧减少的同时，由于系统具有松散耦合的特征使得维护成本也大大减少。



4.3.16 面向服务架构风格

- 优点
 - (1) 灵活性，根据需求变化，重新编排服务。
 - (2) 对IT资产的复用。
 - (3) 使企业的信息化建设真正以业务为核心。业务人员根据需求编排服务，而不必考虑技术细节。

4.3.16 面向服务架构风格

- 缺点
 - (1) 服务的划分很困难。
 - (2) 服务的编排是否得当。
 - (3) 如果选择的接口标准有问题，如主流的 Web service之类，会带来系统的额外开销和不稳定性。
 - (4) 对IT硬件资产还谈不上复用。
 - (5) 目前主流实现方式接口很多，从而很松散很脆弱。
 - (6) 目前主流实现方式只局限于不带界面的服务的共享。

4.3.16 面向服务架构风格

- 应用实例
 - 金蝶EAS

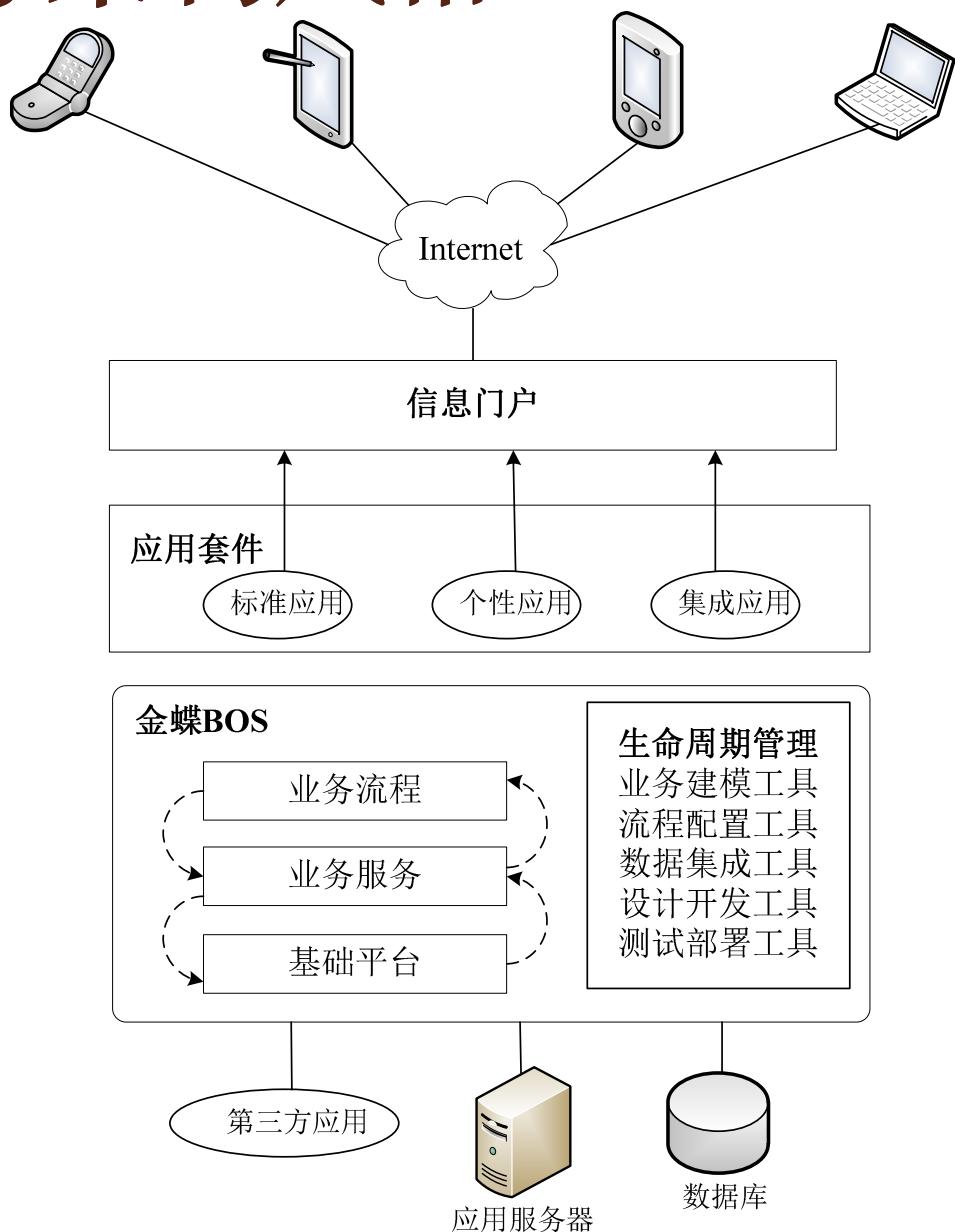


图4.42 金蝶EAS技术架构

4.3.17 正交架构风格

- 正交软件架构(Orthogonal Software Architecture) 由组织层(Layer)和线索(Thread)的组件(Component)构成。
 - 层是由一组具有相同抽象级别(Level of Abstraction)的组件构成。
 - 线索是子系统的特例，它是由完成不同层次功能的组件组成(通过相互调用来关联)，每一条线索完成整个系统中相对独立的一部分功能。
 - 如果线索是相互独立的，即不同线索中的组件之间没有相互调用，那么这个结构就是完全正交的。



4.3.17 正交架构风格

- 正交软件架构是一种以垂直线索组件族为基础的层次化结构
- 其基本思想是把应用系统的结构按功能的正交相关性，垂直分割为若干个线索（子系统），线索又分为几个层次，每个线索由多个具有不同层次功能和不同抽象级别的组件构成。

4.3.17 正交架构风格

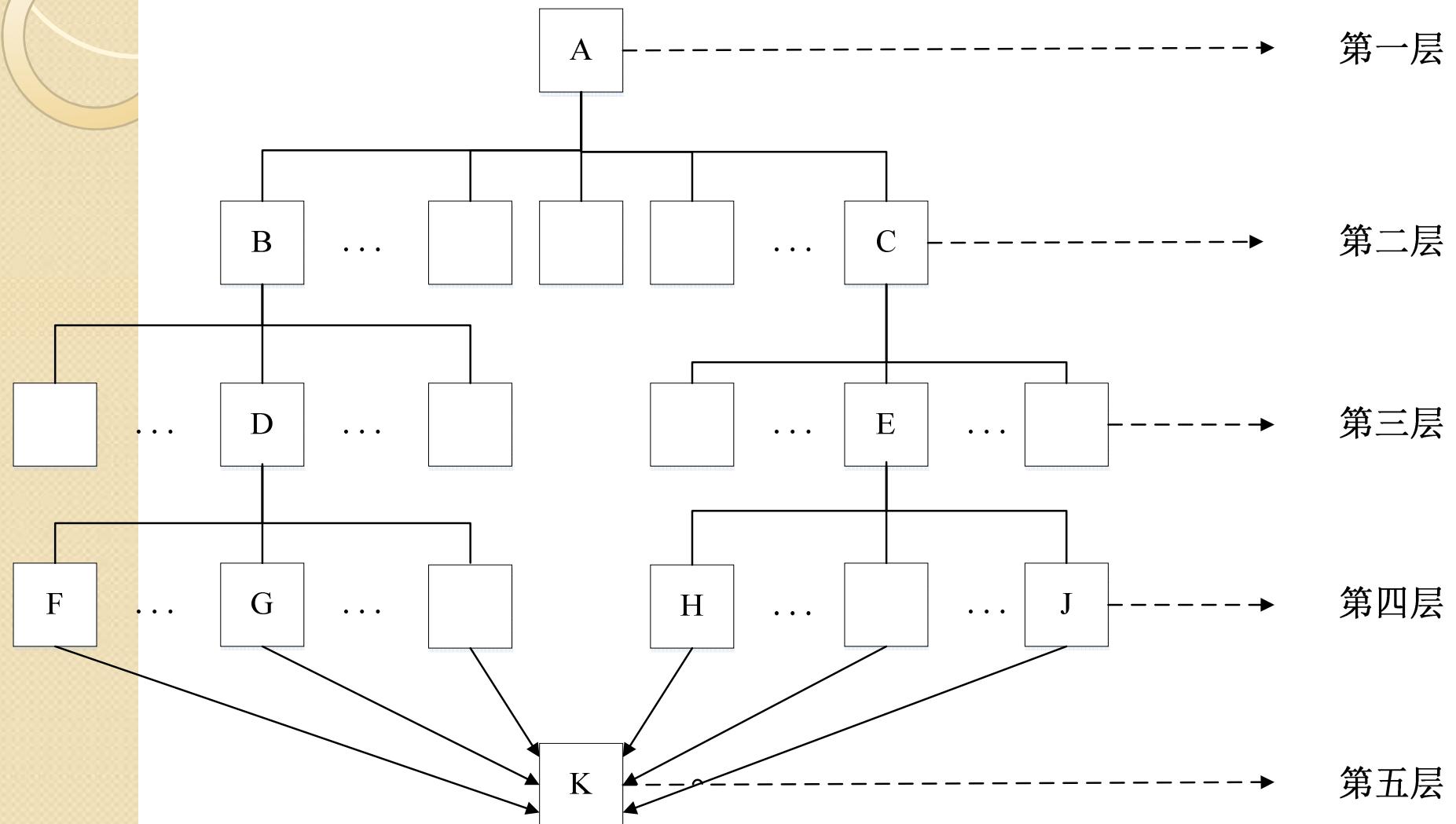


图4.43 正交软件架构

4.3.17 正交架构风格

- 特点：
 - (1) 由完成不同功能的n($n > 1$)个线索(子系统)组成；
 - (2) 系统具有m($m > 1$)个不同抽象级别的层；
 - (3) 线索之间是相互独立的(正交的)；
 - (4) 系统有一个公共驱动层(一般为最高层)和公共数据结构(一般为最低层)。



4.3.17 正交架构风格

- 优点
 - (1) 结构清晰，易于理解。由于线索功能相互独立，组件的位置可以清楚地说明它所实现的抽象层次和负担的功能。
 - (2) 易修改，可维护性强。由于线索之间是相互独立的，所以对一个线索的修改不会影响到其他线索。
 - (3) 可移植性强，重用粒度大。因为正交结构可以为一个领域内的所有应用程序所共享，这些软件有着相同或类似的层次和线索，可以实现架构级的重用。



4.3.17 正交架构风格

- 缺点
 - 在实际应用中，并不是所有软件系统都能完全正交化，或者有时完全正交化的成本太高。因此，在进行应用项目的软件架构设计时，必须反复权衡进一步正交化的额外开销与所得到的更好的性能之间的关系。

4.3.17 正交架构风格

- 应用实例：汽修服务管理系统

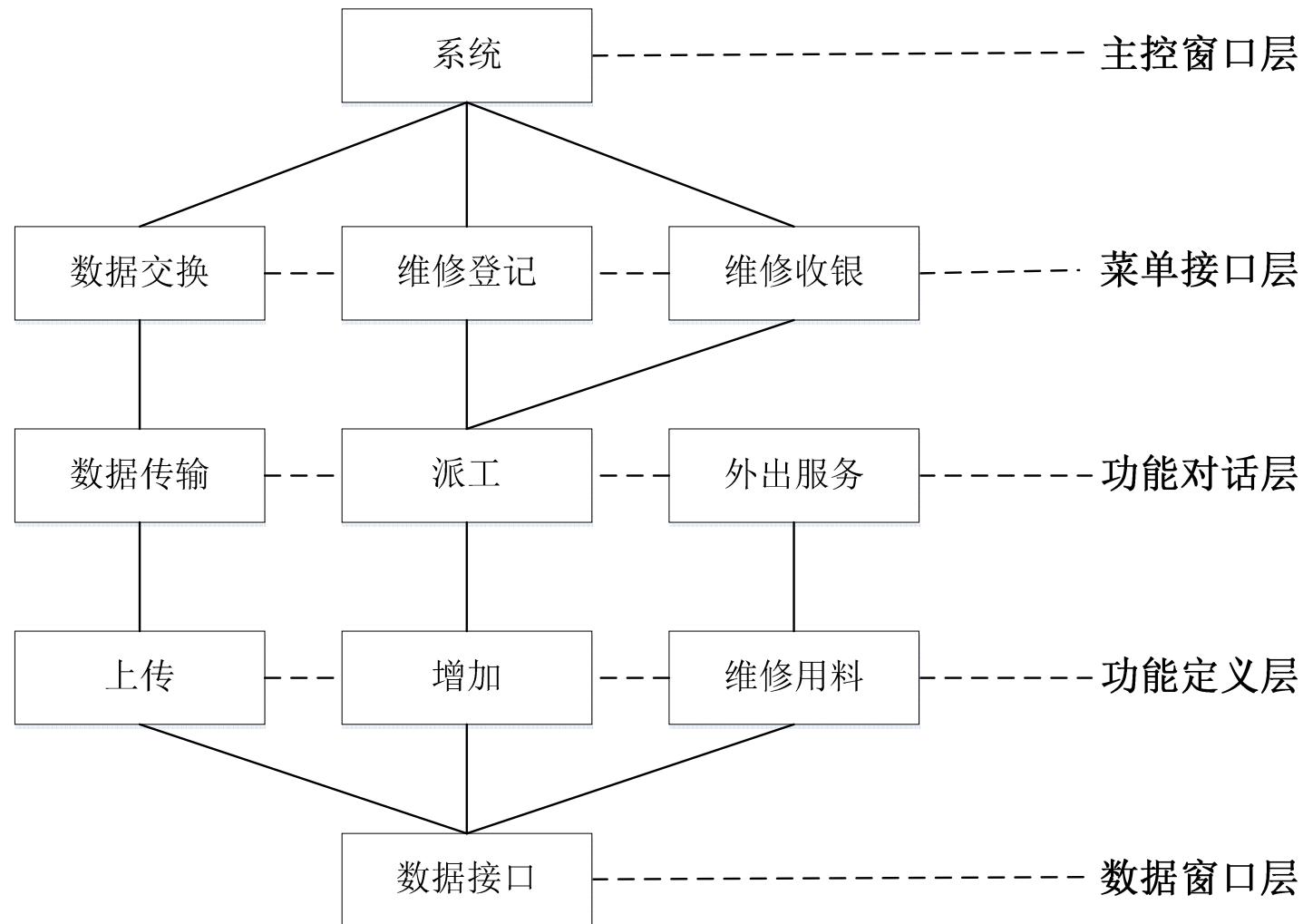


图4.44汽修服务管理系统的框架结构



4.3.18 异构风格

- 在设计软件系统时，从不同角度来观察和思考问题，会对架构风格的选择产生影响。
- 每一种架构风格都有不同的特点，适用于不同的应用问题，因此，架构风格的选择是多样化的和复杂的。
- 在实际应用中，各种软件架构并不是独立存在的，在一个系统中，往往会有多种架构共存和相互融合，形成更复杂的框架结构，即异构架构。



4.3.18 异构风格

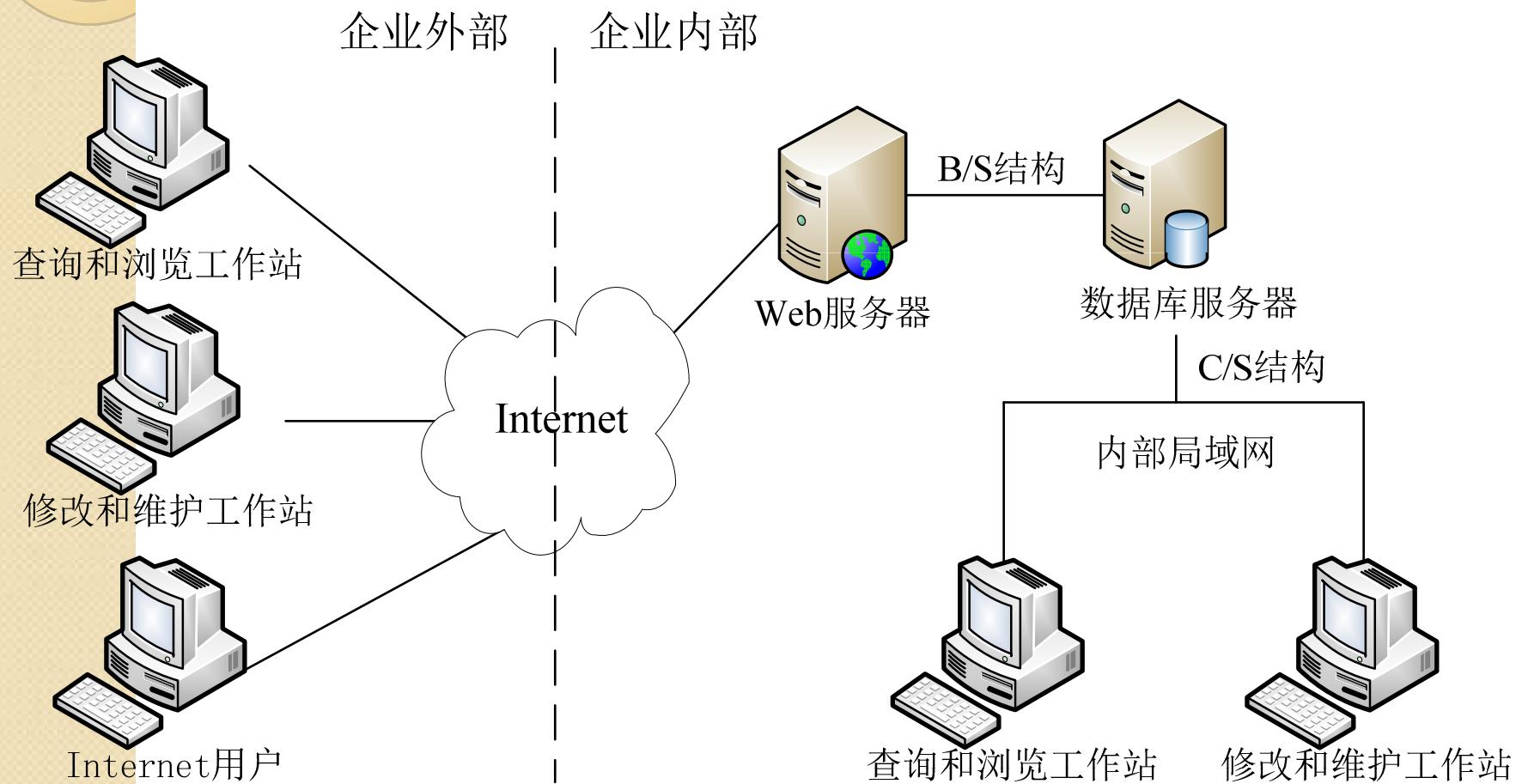
- 异构架构是几种风格的组合，组合方式可能有如下几种：
 - (1) 使用层次结构。一个系统组件被组织成某种架构风格，但它的内部结构可能是另一种完全不同的风格。
 - (2) 允许单一组件使用复合的连接件。

4.3.18 异构风格

- 优点
 - (1) 选择异构架构风格，可以实现遗留代码的重用。
 - (2) 在某一单位中，规定了共享软件包和某些标准，但仍会存在解释和表示习惯上的不同。选择异构架构风格，可以解决这一问题。
- 缺点
 - 不同风格之间的兼容问题有时很难解决

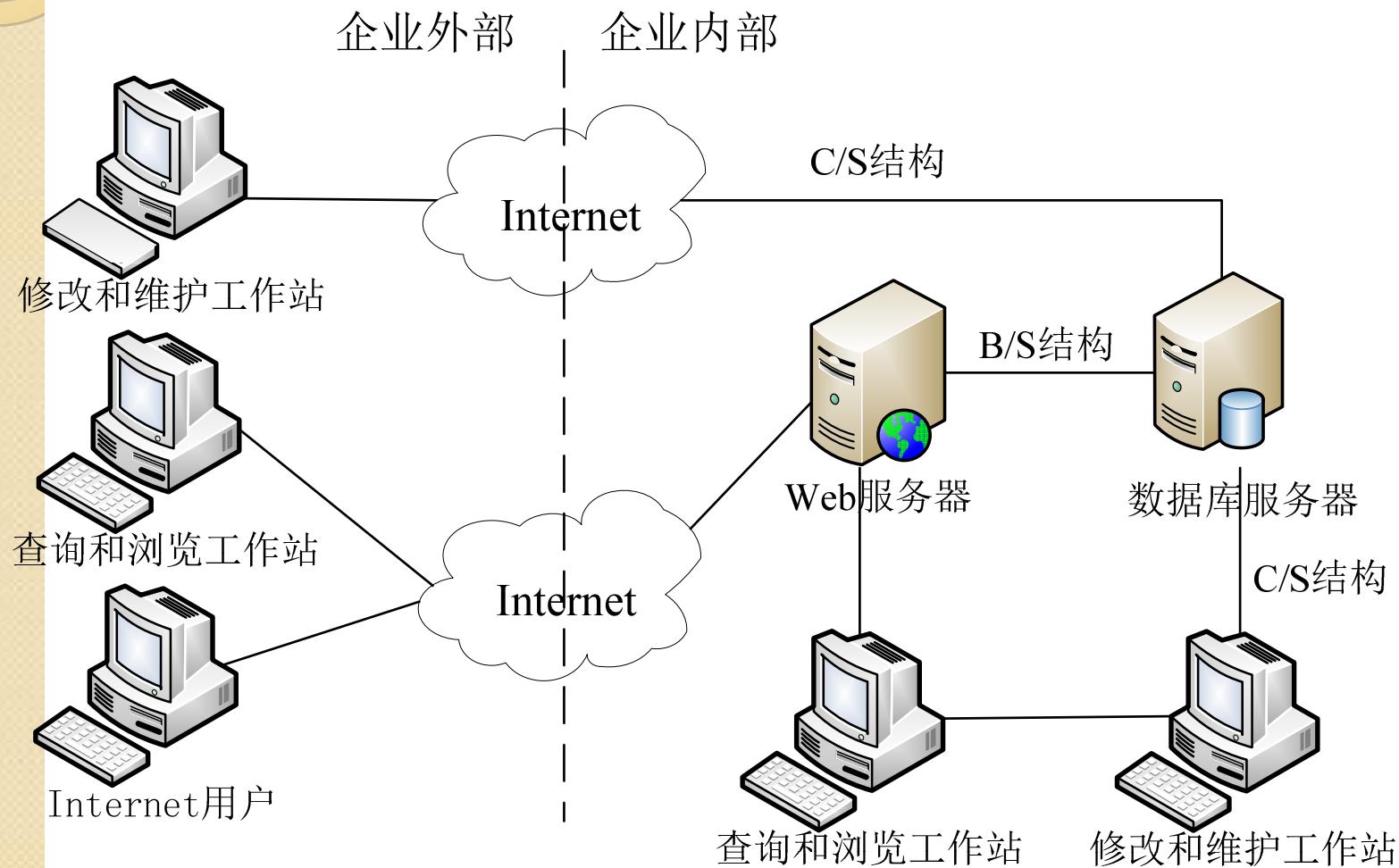
4.3.18 异构风格

- 应用实例：B/S结构和C/S结构组合——“内外有别”模型



4.3.18 异构风格

- 应用实例：B/S结构和C/S结构组合——“查改有别”模型



4.3.18 异构风格

- 应用实例：供电管理系统——B/S结构和C/S结构组合

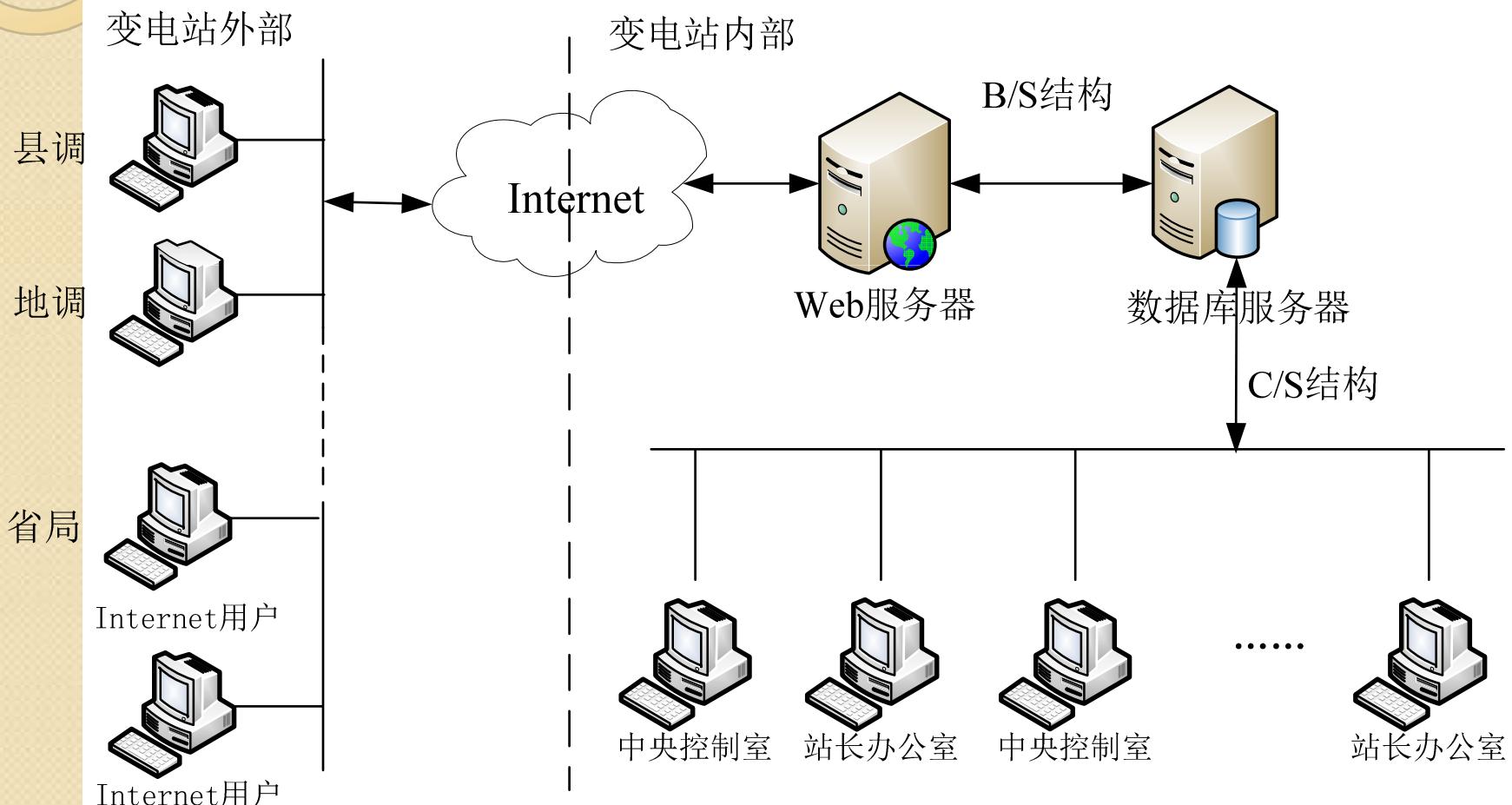


图4.47供电管理系统的架构

4.3.19 基于层次消息总线的架构风格 (JB/HMB风格)

- 以青鸟软件生产线的实践为背景，提出了基于层次消息总线的软件架构(Jade bird hierarchical message bus based style)
- JB/HMB风格基于层次消息总线、支持组件的分布和并发，组件之间通过消息总线进行通讯。
- 消息总线是系统的连接件，负责消息的分派、传递和过滤以及处理结果的返回。各个组件挂接在消息总线上，向总线登记感兴趣的消息类型。

4.3.19 基于层次消息总线的架构风格 (JB/HMB风格)

- 基本思想

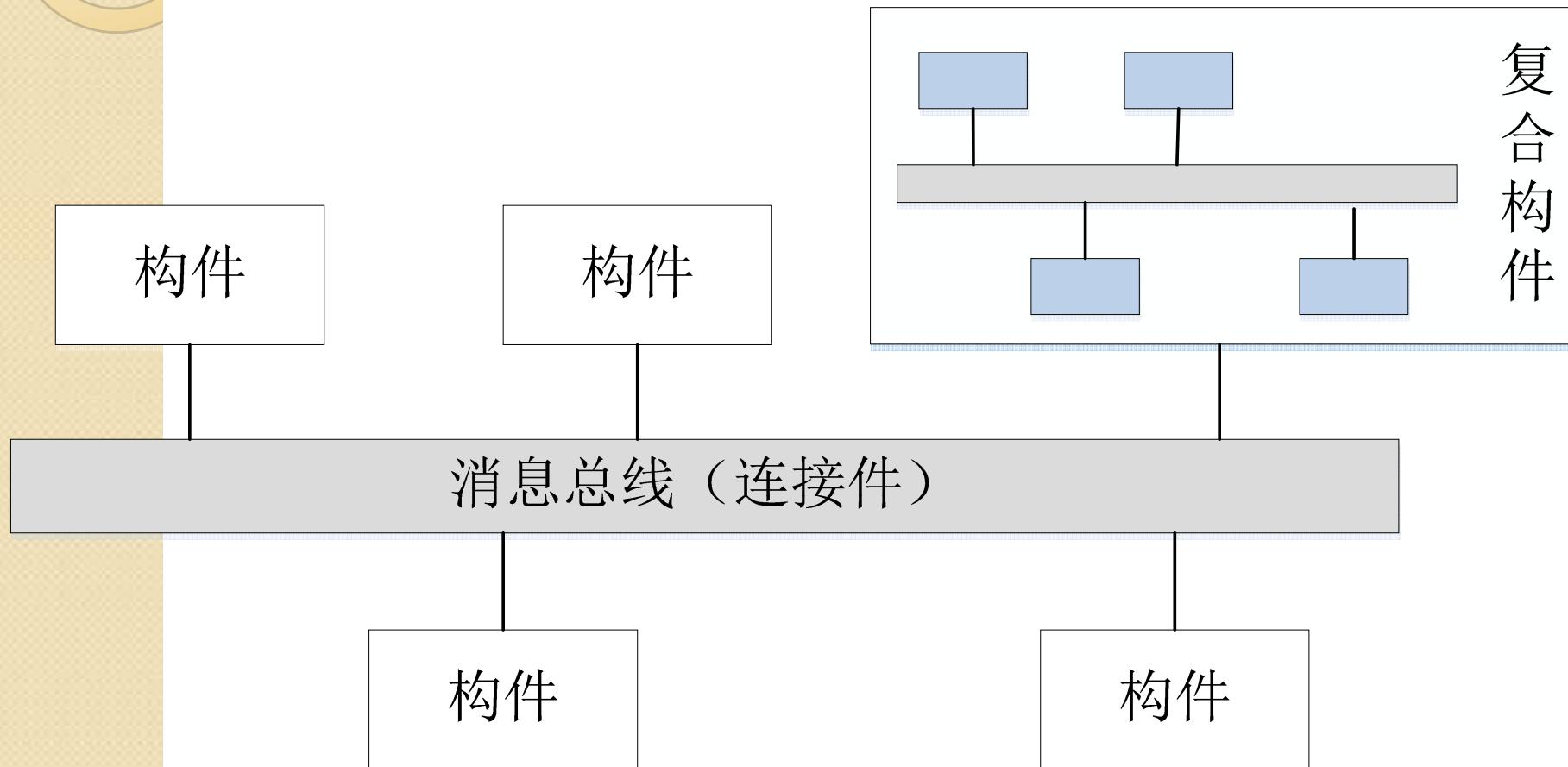


图4.48基于层次消息总线的架构风格

4.3.19 基于层次消息总线的架构风格

- 优点
 - (1) JB/HMB风格的构件接口是一种基于消息的互联接口，可以较好地支持架构设计。降低了构件之间的耦合性，增强了构件的重用性。
 - (2) JB/HMB风格支持运行时系统演化，主要体现在可动态增加和删除构件，动态改变构件所响应的消息以及消息过滤这三个方面。
- 缺点是重用要求高，可重用性差。

4.3.19 基于层次消息总线的架构风格

• 应用实例：青鸟工程软件开发过程

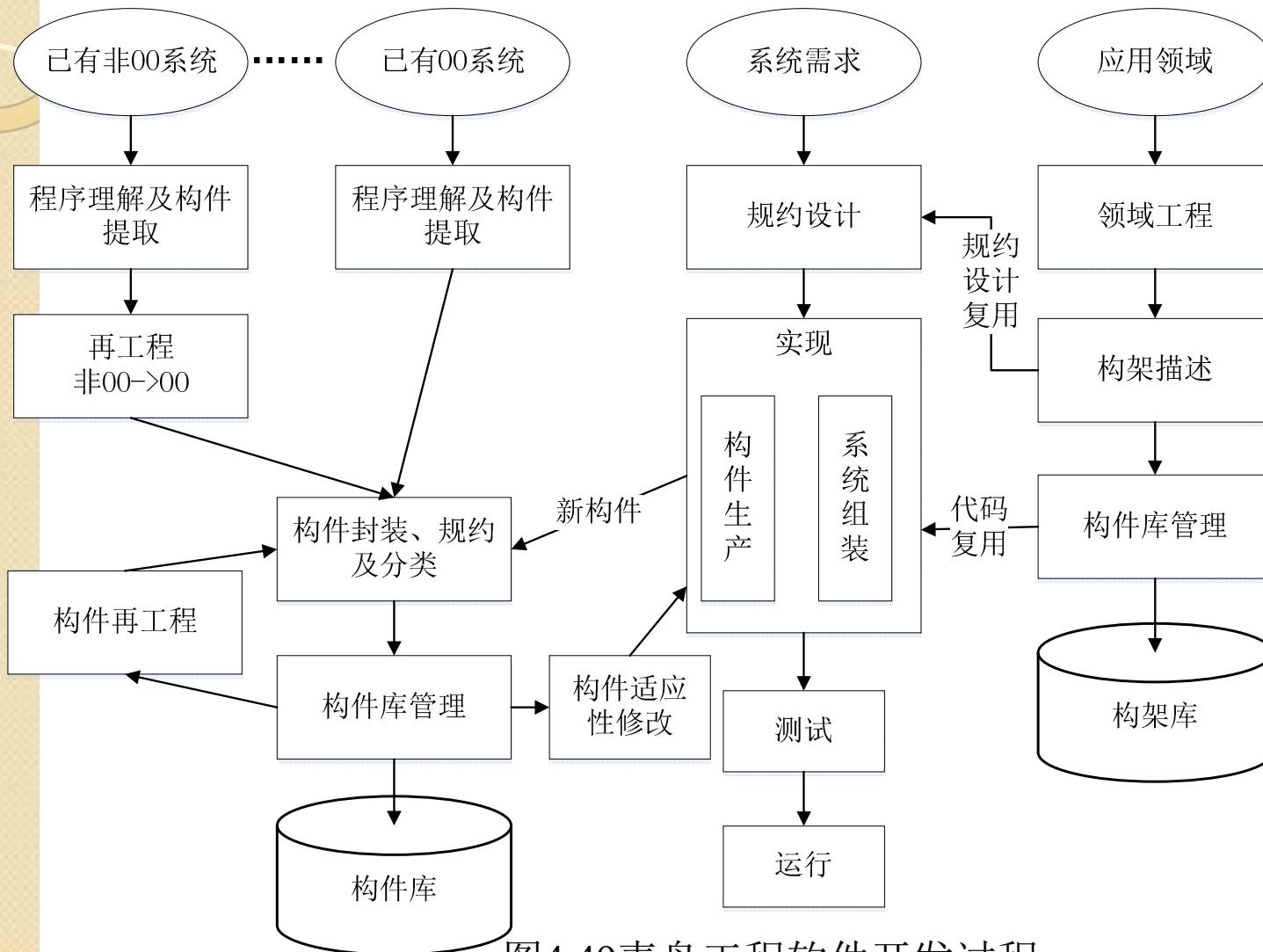


图4.49青鸟工程软件开发过程



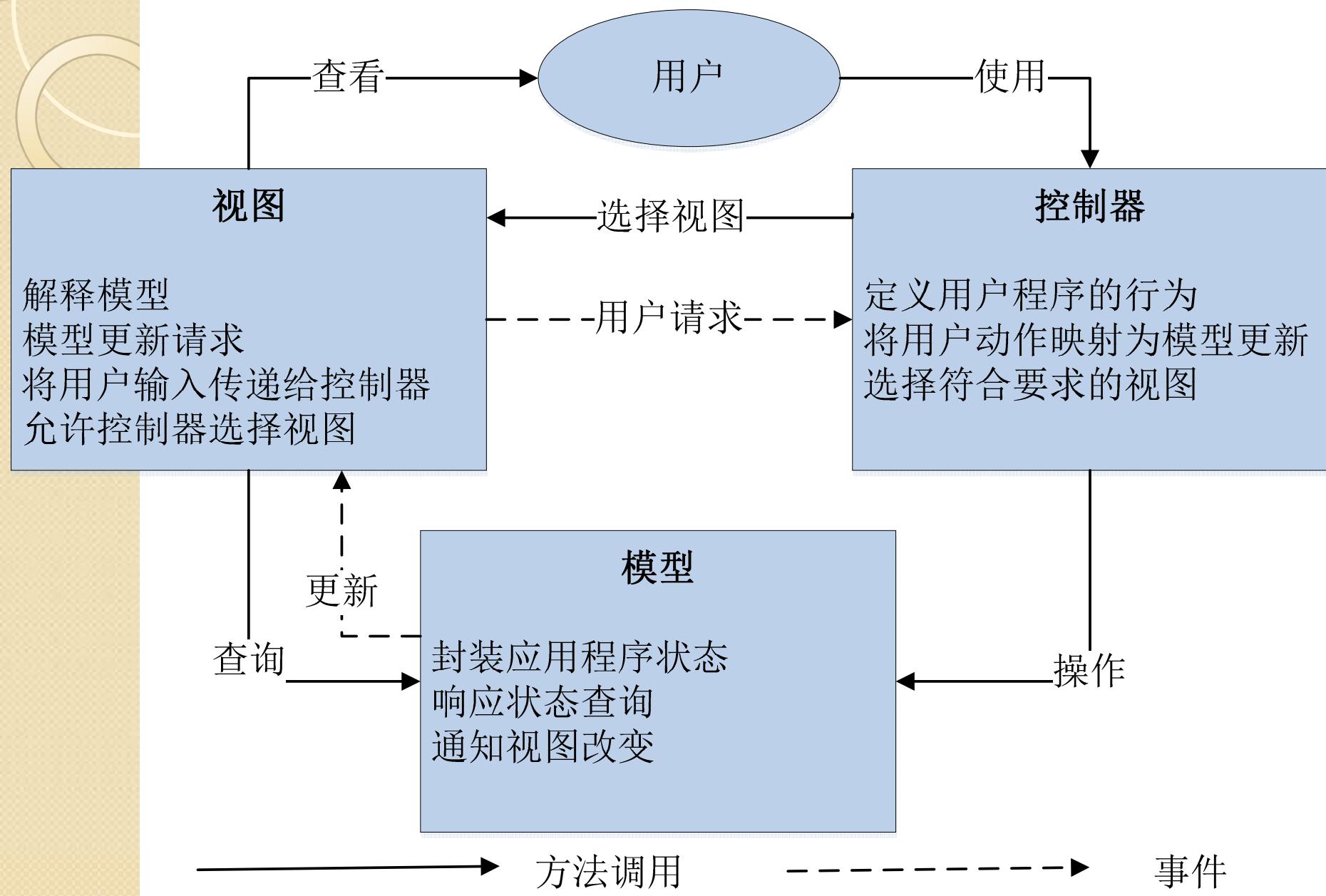
4.3.20 模型-视图-控制器风格

- 模型-视图-控制器风格 (Model-View-Controller, MVC) 主要是针对编程语言Smalltalk 80所提出的一种软件设计模式。
- MVC被广泛的应用于用户交互程序的设计中。

4.3.20 模型-视图-控制器风格

- MVC结构主要包括模型、视图和控制器三部分。
 - (1) 模型(Model, M): 模型是应用程序的核心，它封装了问题的核心数据、逻辑关系和计算功能，提供了处理问题的操作过程。
 - (2) 视图(View, V): 视图是模型的表示，提供了交互界面，为用户显示模型信息。
 - (3) 控制器(Controller, C): 控制器负责处理用户与系统之间的交互，为用户提供操作接口。

4.3.20 模型-视图-控制器风格



4.3.20 模型-视图-控制器风格

- 优点

- (1) 多个视图与一个模型相对应。变化——传播机制确保了所有相关视图都能够及时地获取模型变化信息，从而使所有视图和控制器同步，便于维护。
- (2) 具有良好的移植性。由于模型独立于视图，因此可以方便的实现不同部分的移植。
- (3) 系统被分割为三个独立的部分，当功能发生变化时，改变其中的一个部分就能满足要求。

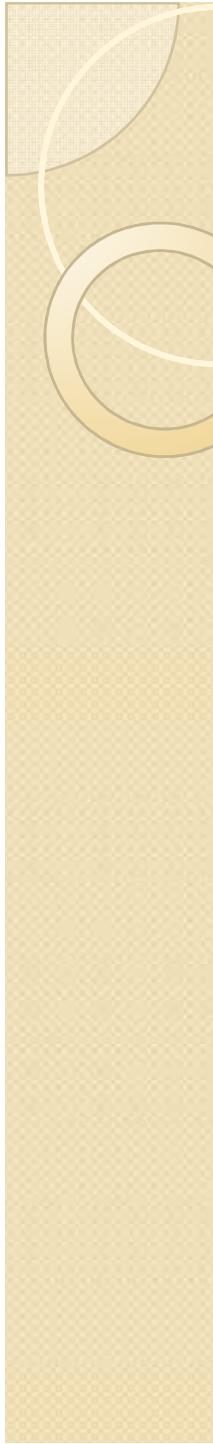
4.3.20 模型-视图-控制器风格

- 缺点

- (1) 增加了系统设计和运行复杂性。
- (2) 视图与控制器连接过于紧密，妨碍了二者的独立重用。
- (3) 视图访问模型的效率比较低。由于模型具有不同的操作接口，因此视图需要多次访问模型才能获得足够的数据。
- (4) 频繁访问未变化的数据，也将降低系统的性能。

4.3.20 模型-视图-控制器风格

- 应用实例
 - 目前比较好的MVC
 - 老牌的有Struts, Webwork。
 - 新兴的MVC框架有Spring MVC, Tapestry, JSF等。
 - 还有，如Dinamica, VRaptor等
 - 这些框架都提供了较好的层次分隔能力。在实现良好的MVC分隔的基础上，提供一些现成的辅助类库，同时也促进了生产效率的提高。



4.4 软件架构模式

- Dwayne E. Perry和Alexander L. Wolf从组件的角度给出了软件架构模式的定义：根据系统的结构组织定义了软件系统族，以及构成系统族的组件之间的关系。



4.4 软件架构模式

- 在一般意义上，大多数人认为模式即为风格
- Frank Buschmann 认为体系结构的模式与风格是有区别的，它们的区别和联系主要在于：
 - (1) 体系结构风格主要描述应用系统的总体结构框架。
 - (2) 架构风格相对独立。
 - (3) 模式比架构风格更加面向问题。